a way in competitive programming

ทศพร แสงจ้า

แนวทางเริ่มเขียนโปรแกรมเชิงแข่งขัน

Contents

1. บทนำ

จะไล่เนื้อหาจากที่เคยสอนในสอวน. ค่าย 1, 2, ผู้แทนศูนย์ และ สสวท.

โดยเนื้อหาในแต่ละช่วงจะเป็นประมาณว่า

- ค่ายหนึ่ง: เริ่มเขียนภาษา C/C++ เขียนยังไงให้คอมไพล์ผ่าน ถ่ายทอดความคิดออกมาเป็นโค้ดให้ได้
- ค่ายสอง: เริ่มเขียนโปรแกรมเชิงแข่งขัน เขียนยังไงให้เร็ว วิเคราะห์อัลกอริทึม
- ต่อไป: เนื้อหาที่ซับซ้อนมากขึ้นตามระดับค่าย

โดยเนื้อหาเรียงตามนี้

- ค่ายหนึ่ง
  - template โปรแกรม และ syntax
    - common function ใช้บ่อย
  - concept พื้นฐาน
    - if-else
    - loop
    - function
    - recursion
- ค่ายสอง
  - การเขียนโปรแกรมเชิงแข่งขัน
  - data structure
  - algorithm

2. ค่ายหนึ่ง

2.1. พื้นฐาน

2.1.1. โปรแกรมที่ใช้เขียน

- เริ่มด้วย VS Code[1] กับ plugin cph[2] น่าจะง่ายสุดแล้ว

2.1.2. โค้ดเริ่มต้น

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  return 0;
}
```

- `bits/stdc++.h` เป็น library ครอบจักรวาล
- `using namespace std` ใช้เพื่อความสะดวก
  - เดิมต้องเขียน `std::cout << "Hello World!";` ในการแสดงผล `Hello World!`
  - ถ้ามี `using namespace std` จะไม่ต้องพิมพ์ `std::`
  - ก็คือจะเหลือ `cout << "Hello World!;` พอ

2.1.3. Comments

คอมเม้นท์ไว้บอกคอมว่าไม่ต้องรันบรรทัดนั้นๆ ทำได้สองแบบ

```cpp
// line comment

/*
block
comment
*/
```

2.1.4. Variables Data Types

ในภาษา C++ จำเป็นจะต้องประกาศ “ชนิด” ของตัวแปรเสมอ (ต่างจาก Python)

ชนิดที่พบบ่อยคือ

ชนิด	ขนาด (byte)	
`bool`	1	ค่าความจริง true, false (0-1)
`char`	2	ตัวอักษร
`int`	4	จำนวนเต็ม
`long long`	8	จำนวนเต็มที่เก็บค่าได้มากกว่า
`float`	4	จำนวนจริง

[1] https://code.visualstudio.com/°

[2] https://marketplace.visualstudio.com/items?itemName=DivyanshuAgrawal.competitive-programming-helper°

ชนิด	ขนาด (byte)	
`double`	8	จำนวนจริงที่เก็บค่าได้มากกว่า

สาเหตุที่มีชนิดเดียวกันที่เก็บค่าได้แตกต่างกันเพราะการจองพื้นที่ memory ขนาดไม่เท่ากัน

2.1.5. Input/Output

- ใช้ `cin` ในการรับค่า โดยที่ `cin` จะดูชนิดของตัวแปรให้ตอนที่รับค่า

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  int a;
  cin >> a;
  return 0;
}
```

หากพิมพ์ `5` หลังจากรันโปรแกรมแล้ว `a` จะมีค่าเท่ากับ 5 ที่เป็น `integer`

ถ้า

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  char a;
  cin >> a;
  return 0;
}
```

แล้วพิมพ์ `5` หลังจากรันโปรแกรมแล้ว `a` จะมีค่าเท่ากับ `'5'` ที่เป็น `character`

ตัวอย่างเช่น

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  char a;
  int b;
  cin >> a >> b;
  int c = a;
  cout << c << " " << b;
  return 0;
}
```

แล้วพิมพ์ `5 5` จะได้ผลลัพธ์เป็น `53 5` เนื่องจาก character `'5'` มีค่า ascii เป็น 53 (ถูกแปลงค่า)

- `'\n'` เป็น special character ที่ใช้สำหรับขึ้นบรรทัดใหม่ในการแสดงผล เช่น

```cpp
#include <bits/stdc++.h>
```

```cpp
using namespace std;

int main() {
  char a;
  int b;
  cin >> a >> b;
  int c = a;
  cout << c << "\n" << b;
  return 0;
}
```

แล้วพิมพ์ `5 5` จะได้ผลลัพธ์เป็น

```
53
5
```

- สังเกตว่า `cout` ต้องใช้เป็นท่อน ๆ คือส่ง `<<` ได้ทีละตัวแปรหรือทีละ string เท่านั้น
- เช่นเดียวกัน `cin` ที่ใช้เป็นท่อน ๆ คือรับ `>>` เข้าทีละตัวแปร โดยตัด input ตามประเภทตัวแปร

2.1.6. Control Structures

2.1.6.1. condition

- `if` - `else if` - `else`

```cpp
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

2.1.6.2. loop

- `while`

```cpp
int i = 0;
while (i < 10) {
  cout << i << " ";
  i++;
}
```

- `for`

```cpp
for (int i = 0; i < 10; i++) {
  cout << i << " ";
}
```

2.1.7. Arrays

ใน C++ นั้นตัวแปรทุกชนิดสามารถทำเป็นตัวแปรชุด (Array) ได้ด้วยการใส่วงเล็บ `[]` ด้านหลังชื่อตัวแปร เช่น

จาก `int a` เป็นการประกาศตัวแปรที่เป็นจำนวนเต็มหนึ่งจำนวน

สามารถเป็น `int a[10]` เป็นการประกาศตัวแปรที่เป็นจำนวนเต็ม `10` จำนวน โดยจะเรียกสมาชิกได้ เช่น

`a[0]` เป็นการเรียกสมาชิกตัวแรกใน array นั้น

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  int n;
  int a[20];
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> a[i];
  }
  return 0;
}
```

ตัวอย่างด้านบนจะเป็นการรับค่า `n` เพื่อที่จะรับจำนวนเต็มต่ออีก n จำนวน (ใช้ต่อในเงื่อนไขของ for loop)

จากนั้นรับค่าเข้าสู่ `a[i]` ไปเรื่อย ๆ โดย $i = \{0, 1, ..., n\}$

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  int n;
  int a[20];
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> a[i];
  }
  int sm = 0;
  for (int i = 0; i < n; i++) {
    sm += a[i];
  }
  cout << sm;
  return 0;
}
```

ตัวอย่างด้านบนเป็นการนำค่าที่รับเข้ามาไปใช้หาผลรวมต่อในตัวแปร `sm`

2.1.8. Strings

ใน C++ มี `string` ให้ใช้ใน STL ซึ่งถูกรวมอยู่ใน `bits/stdc++.h` แล้ว

จากที่ภาษา C ธรรมดาจะต้องใช้เป็น array of characters จึงสะดวกขึ้นเยอะ

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  string a;
  cin >> a;
  cout << a;
```

```
  return 0;
}
```

ตัวอย่างด้านบนเป็นการรับค่า string แล้วส่งออกเลย

2.1.9. Functions

```
#include <bits/stdc++.h>

using namespace std;

int addint(int a, int b) {
  return a + b;
}

int main() {
  cout << addint(1, 4);
  return 0;
}
```

ตัวอย่างด้านบนเป็นการสร้างฟังก์ชัน $f(a, b) = a + b$ โดยให้ชื่อว่า `addint` พร้อมตัวอย่างการเรียกใช้

2.1.10. Array หลายมิติ

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  int a[5][5];
  int n, m;
  cin >> n >> m;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      cin >> a[i][j];
    }
  }
  return 0;
}
```

2.2. STL library ที่ใช้บ่อย

2.2.1. Max/Min

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  cout << max(2, 4); // 4
  cout << min(2, 4); // 2
  return 0;
}
```

2.2.2. Sort

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;

int main() {
  int a[] = {3, 1, 2};
  cout << a[0] << " " << a[1] << " " << a[2] << "\n"; // 3 1 2
  sort(a, a+3);
  cout << a[0] << " " << a[1] << " " << a[2] << "\n"; // 1 2 3
  return 0;
}
```

2.2.3. Vector

- เป็น dynamic array (array ที่สามารถเพิ่มลดขนาดได้)
- `vector` มีการใช้งานโดยใช้ `vector<T> varName;` เช่น `vector<int> A;`

2.2.3.1. Access

- `A[i]` ได้เหมือน array
- `A.front()` และ `A.back()`

2.2.3.2. Functions

- `A.size()` เช็คขนาดของ vector
- `A.empty()` เช็คว่า vector ว่างหรือไม่
- `A.push_back(10)` ไว้เพิ่มสมาชิกต่อท้าย
- `A.pop_back()` ไว้ลบสมาชิกตัวสุดท้าย

2.2.3.3. Loop

```cpp
for (int i = 0; i < A.size(); i++)
  cout << A[i] << endl;
```

2.2.3.4. Example

```cpp
#include <bits/stdc++.h>

using namespace std;

void printVec(vector<int> v) {
  for (int i = 0; i < v.size(); i++) {
    cout << v[i] << " ";
  }
  cout << "\n";
}

int main() {
  vector<int> A;
  A.push_back(2);
  A.push_back(5);
  printVec(A); // OUTPUT: 2 5 \n

  A.push_back(9);
  printVec(A); // OUTPUT: 2 5 9 \n

  cout << A.front() << "\n"; // OUTPUT: 2\n
```

```
  cout << A.back() << "\n";  // OUTPUT: 9\n
  cout << A.size() << "\n";  // OUTPUT: 3\n

  A.pop_back();
  A.pop_back();
  printVec(A); // OUTPUT: 2 \n
}
```

2.2.4. Pair

- เป็น struct สำเร็จรูปที่มีสมาชิกสองตัว
- `pair<T1, T2> varName;` เช่น `pair<int, double> A;`

2.2.4.1. Access

- `A.first` และ `A.second`

2.2.4.2. Example

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  pair<int, double> A;
  A.first = 4;
  A.second = 7.5;
  cout << A.first << "\n";  // OUTPUT: 4
  cout << A.second << "\n"; // OUTPUT: 7.5
  A = pair<int, double>(6, 2.3);
  cout << A.first << "\n";  // OUTPUT: 6
  cout << A.second << "\n"; // OUTPUT: 2.3
}
```

2.3. Function

- ตัวอย่างฟังก์ชันที่มีใน `bits/stdc++.h`
- วิธีการเขียน user-defined functions
- เทียบมุมมองของฟังก์ชันทางคณิตศาสตร์และทางคอมพิวเตอร์

2.3.1. แนวคิด

การเขียนฟังก์ชันในโปรแกรมเป็นการรวมกลุ่มคำสั่งเพื่อทำงานตามจุดประสงค์ โดยสามารถเรียกใช้งานภายหลังในโปรแกรมนั้นได้

2.3.2. การเขียนโปรแกรม

2.3.2.1. ตัวอย่างฟังก์ชันพร้อมเรียกใช้งาน

ใน C++ Standard Template Library (STL) จะมีฟังก์ชันพร้อมเรียกใช้งานให้ใช้ ซึ่งเมื่อ

```
#include <bits/stdc++.h>
```

แล้ว ในการเขียนโปรแกรมเชิงแข่งขันจะทำการ include STL ที่จำเป็นทั้งหมดมาให้ ซึ่งตัวอย่างฟังก์ชันที่ใช้งานบ่อยครั้งมีดังนี้

- abs, pow, sqrt, log2, exp จาก cmath

- swap, min, max, __gcd, next_permutation จาก algorithm

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
  cout << "cmath examples\n";
  cout << "abs(-5): " << abs(-5) << "\n";
  cout << "pow(2, 3): " << pow(2, 3) << "\n";
  cout << "sqrt(64): " << sqrt(64) << "\n";
  cout << "log2(64): " << log2(64) << "\n";
  cout << "exp(3): " << exp(3) << "\n";
  cout << "\n";

  cout << "algorithm examples\n";
  int a = 4, b = 7;
  cout << "a b: " << a << " " << b << "\n";
  swap(a, b);
  cout << "a b: " << a << " " << b << "\n";
  cout << "min(a, b): " << min(a, b) << "\n";
  cout << "max(a, b): " << max(a, b) << "\n";
  cout << "__gcd(a, b): " << __gcd(a, b) << "\n";
}
```

Output:

```
cmath examples
abs(-5): 5
pow(2, 3): 8
sqrt(64): 8
log2(64): 6
exp(3): 20.0855

algorithm examples
a b: 4 7
a b: 7 4
min(a, b): 4
max(a, b): 7
__gcd(a, b): 1
```

2.3.2.2. User-defined functions

เราสามารถเขียนฟังก์ชันด้วยตนเองได้ โดยการระบุ:
- ชนิด (type) ของค่าที่จะคืนค่า
- ชื่อ (name) ที่ใช้เรียกฟังก์ชันนั้น
- พารามิเตอร์ (parameters) ที่ไว้รับค่าเข้าสู่ฟังก์ชัน
- ชุดคำสั่ง (statements) ที่ระบุวิธีการทำงานของฟังก์ชันนั้น

ในรูปแบบ:

```
type name ( param1, param2, ...) { statements }
```

ตัวอย่างการเขียนฟังก์ชัน:

```cpp
#include <bits/stdc++.h>
using namespace std;
```

```
int addition(int a, int b) {
  int r;
  r = a + b;
  return r;
}

int main() {
  int z;
  z = addition(5, 3);
  cout << "The result is " << z;
}
```

Output:

```
The result is 8
```

2.3.2.3. มุมมองของฟังก์ชัน

ทางคณิตศาสตร์จะเขียนฟังก์ชันอยู่ในรูป $y = f(x)$ ซึ่งเมื่อเทียบกับทางคอมพิวเตอร์แล้วจะเทียบเท่าเป็นฟังก์ชันที่มี:

- ไม่ได้ระบุชนิดค่าที่จะคืนออกมา
- ชื่อ f
- รับพารามิเตอร์ x

ตัวอย่างฟังก์ชันเอกลักษณ์:

$f(x) = x$

ในภาษา C++:

```
int f(int x) {
  return x;
}
```

2.3.3. โจทย์ตัวอย่าง - ค่าถัดไปของฟังก์ชันเวียนเกิด

2.3.3.1. เนื้อหาโจทย์

จงเขียนฟังก์ชัน `next_a(int p, int pn)` หาค่า:

$a_n = a_{\{n-1\}} + n \times 2$

เมื่อทราบค่า $a_{\{n-1\}}$

2.3.3.2. รหัสเทียม

```
int next_a(int p, int pn) {
    return p + (pn + 1) * 2;
}
```

2.3.4. โจทย์ตัวอย่าง - ระยะห่างระหว่างสองจุด

2.3.4.1. เนื้อหาโจทย์

กำหนดให้ $A(x_1, y_1)$ และ $B(x_2, y_2)$ เป็นจุดสองจุดบนระนาบ

2.3.4.2. รหัสเทียม

```
double distance(double x1, double y1, double x2, double y2) {
    return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}
```

2.3.5. โจทย์ตัวอย่าง - ระยะห่าง Manhattan

2.3.5.1. เนื้อหาโจทย์

จงเขียนฟังก์ชัน `manhattan(x1, y1, x2, y2)` เพื่อหาระยะ Manhattan ระหว่าง A และ B

2.3.5.2. รหัสเทียม

```
double manhattan(double x1, double y1, double x2, double y2) {
    return abs(x1 - x2) + abs(y1 - y2);
}
```

2.3.6. ปัญหาที่พบบ่อย

2.3.6.1. Scope of variables

แต่ละฟังก์ชันมี scope ของตัวเอง ตัวแปรแบ่งเป็น:

- Global variables
- Local variables

ตัวอย่าง:

```
#include <bits/stdc++.h>
using namespace std;

int g;

int addition(int a, int b) {
  int r;
  g = b;
  r = a + b;
  return r;
}

int main() {
  int z;
  z = addition(5, 3);
  cout << "z is " << z << "\n";
  cout << "g is " << g;
}
```

Output:

```
z is 8
g is 3

Errors when accessing out-of-scope variables:

cout << "z is " << z << "\n";
```

2.3.6.2. Return/type

ลืม return จะเกิด warning

```
#include <bits/stdc++.h>
using namespace std;

int addition(int a, int b) {
  int r;
  r = a + b;
}
```

Output:

```
warning: no return statement in function returning non-void
```

2.3.6.3. Arguments passed by value and by references

เพื่อให้เปลี่ยนค่าพารามิเตอร์ใช้ & ข้างหน้า

ตัวอย่าง:

```
void swap(int &a, int &b) {
  int c = a;
  a = b;
  b = c;
}

in main: 0 1
in swap: 1 0
in main: 1 0
```

2.4. Recursion

2.4.1. แนวคิด

- Recursion คือกระบวนการเรียกตัวเอง
- ตามปกติแล้วแบ่งกรณีเป็น
  - base case
  - recursion case

  โดยที่จะย่อยปัญหาให้เล็กลงจนกระทั่งแก้ได้ด้วย base case

2.4.2. การเขียนโปรแกรม

จะเขียนกระบวนการนั้นด้วยฟังก์ชัน และการเรียกตัวเองจะเป็นการเรียกฟังก์ชันที่เขียนขึ้นมาภายในฟังก์ชันนั้น

ตัวอย่างการเขียน recursive function ในภาษา C++ โดยยังไม่คำนึงถึงการนำไปใช้แก้โจทย์ปัญหา

```
void recursive_function() {
    if (base condition)
        return; // base case
    else
        recursive_function(); // recursion case
}
```

2.4.3. วิเคราะห์ประสิทธิภาพ

จะใช้วิธีการนับจำนวนครั้งที่จะมีการเรียกฟังก์ชันเกิดขึ้นตามพารามิเตอร์ของฟังก์ชันนั้น

2.4.4. โจทย์ตัวอย่าง - Factorial

2.4.4.1. เนื้อหาโจทย์

จงหาค่าของ $n!$ โดย $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$

2.4.4.2. แนวคิด

โจทย์นี้สามารถใช้โครงสร้าง loop ในการแก้ปัญหาได้ แต่นำมาเป็นตัวอย่างเริ่มต้นในการใช้โครงสร้างของ recursion ในการแก้ปัญหา

โดยมองค่าของ $n!$ เป็น recursive function ดังนี้

$$f(n) = \begin{cases} 1 \text{ if } n \leq 1 \\ n \times f(n-1) \text{ else} \end{cases} \tag{1}$$

2.4.4.3. รหัสเทียม

```cpp
#include <bits/stdc++.h>

using namespace std;

int factorial(int n) {
  if (n ≤ 1) { // base case
    return n;
  } else {
    return n * factorial(n - 1);
  }
}

int main() {
  cout << factorial(4);
  return 0;
}
```

จะได้ output

```
24
```

2.4.5. โจทย์ตัวอย่าง - Fibonacci

2.4.5.1. เนื้อหาโจทย์

จงหาค่าของเลขฟิโบนัชชีลำดับที่ n โดยที่ เลขฟิโบนัชชีลำดับที่ n จะมีค่าเท่ากับผลบวกของเลขฟิโบนัชชีลำดับที่ $n-1$ และ $n-2$

2.4.5.2. แนวคิด

โจทย์นี้สามารถใช้โครงสร้าง loop ในการแก้ปัญหาได้เช่นกัน แต่นำมาเป็นอีกตัวอย่างหนึ่งในการใช้โครงสร้างของ recursion ในการแก้ปัญหา

โดยมองค่าของเลขฟิโบนัชชีลำดับที่ n เป็น recursive function ดังนี้

$$f(n) = \begin{cases} n \text{ if } n \leq 1 \\ f(n-1) + f(n-2) \text{ else} \end{cases} \tag{2}$$

2.4.5.3. รหัสเทียม

```
#include <bits/stdc++.h>

using namespace std;

int fibo(int n) {
  if (n ≤ 1) { // base case
    return 1;
  } else {
    return fibo(n - 1) + fibo(n - 2);
  }
}

int main() {
  cout << fibo(4);
  return 0;
}
```

จะได้ output

```
3
```

2.4.6. โจทย์ตัวอย่าง - Permutation

2.4.6.1. เนื้อหาโจทย์

จงหาลำดับที่มีสมาชิก n ตัว โดยที่แต่ละสมาชิกมีค่าน้อยกว่า r ทั้งหมดที่เป็นไปได้ (จำเป็นต้องใช้ array)

2.4.6.2. แนวคิด

เป็นการค้นทุกวิธีที่เป็นไปได้ (complete search) ซึ่งจะนำ recursive function เข้ามาใช้ โดยเริ่มต้นด้วยการสร้างลำดับที่ยังไม่มีสมาชิกใด ๆ และค่อย ๆ เติมสมาชิกเข้าไปในลำดับนั้น

base case จะเป็นเงื่อนไขเมื่อลำดับมีสมาชิกครบ n

recursive case จะเป็นการเติมสมาชิกลำดับถัดไปด้วยทุกค่าที่เป็นไปได้

2.4.6.3. รหัสเทียม

```
#include <bits/stdc++.h>

using namespace std;

int a[10];

void perm(int d, int r, int n) {
  if (d == n) { // base case
    for (int i = 0; i < n; i++) {
      cout << a[i] << " ";
    }
    cout << "\n";
    return;
```

```
  }
  for (int i = 0; i < r; i++) {
    a[d] = i;
    perm(d + 1, r, n); // recursive case
  }
}

int main() { perm(0, 3, 3); }
```

จะได้ output

```
0 0 0
0 0 1
...
2 2 2
```

2.4.7. ปัญหาที่พบบ่อย

2.4.7.1. Overflow

กรณีไม่มี base case หรือ base case ไม่ครอบคลุม จะทำให้เกิดการ overflow ได้

```
#include <bits/stdc++.h>

using namespace std;

int factorial(int n) { return n * factorial(n - 1); }

int main() {
  cout << factorial(4);
  return 0;
}
```

จะเกิด error เช่น:

```
"./a.out" terminated by signal SIGSEGV (Address boundary error)
```

2.4.8. โจทย์แนะนำ

- หาลำดับ permutation โดยมีเงื่อนไขที่แตกต่างกัน
  - จำนวน 5 หลัก และแต่ละหลักมีค่าไม่เกิน 6
  - ตัวเลขก่อนหน้าต้องน้อยกว่าหรือเท่ากับตัวถัดไป
  - ตัวเลขห้ามซ้ำ
- กลับข้อความ (reverse string) ห้ามใช้ loop, array หรือ string
  - กลับทั้งข้อความ เช่น: “iamastring” เป็น “gnirtsamai”
  - กลับเฉพาะภายในคำที่คั่นด้วยเว้นวรรค เช่น: “i am a string” เป็น “i ma a gnirts”
- Programming.in.th
  - 0019 Perket
  - 0039 Food
- cses.fi
  - 2205 Gray Code
  - 2165 Tower of Hanoi
  - 1622 Creating Strings

- 1623 Apple Division

3. ค่ายสอง

3.1. การเขียนโปรแกรมเชิงแข่งขัน

3.1.1. หลักการ

การเขียนโปรแกรมเชิงแข่งขันแบ่งออกเป็นสองหัวข้อหลักๆ คือ

3.1.1.1. การออกแบบวิธีการคิด

ส่วนนี้จะเน้นเรื่องทักษะการแก้ปัญหาและพื้นฐานการคิด บางทีต้องมีความคิดสร้างสรรค์ ประกอบกับทฤษฎีต่างๆ โดยทั่วไปแล้ววิธีการแก้ปัญหาจะประกอบด้วยเทคนิคที่รู้กันทั่วไปร่วมกับความเป็นเอกลักษณ์ของปัญหานั้นๆ

3.1.1.2. การเขียนโปรแกรมตามวิธี

ส่วนนี้จะเน้นว่าเราสามารถถ่ายทอดวิธีการคิดของเราสู่โปรแกรมคอมพิวเตอร์ได้อย่างถูกต้อง ปกติแล้วแค่วิธีคิดถูกต้องนั้นไม่เพียงพอในการแข่งขันเขียนโปรแกรมให้ได้ดี ข้อแตกต่างสำคัญกับการเขียนโปรแกรมทั่วไปคือโปรแกรมในการแข่งขันจะสั้น (มากที่สุดหลักไม่กี่ร้อยบรรทัด) และไม่จำเป็นต้องดูแลโปรแกรมหลังจากแข่งเสร็จ

3.1.2. การบริหารเวลา

เป็นสิ่งสำคัญที่สุดในการแข่งขัน

- ให้คำนึงให้ได้ว่าเราจะใช้เวลาเท่าไรในการทำโจทย์ลักษณะนี้ แล้วจะได้คะแนนเท่าไร แล้วลำดับความสำคัญในการเลือกโจทย์ให้ได้
- พยายามสร้างกฎที่เหมาะสมกับตัวเอง เช่น ถ้าบัคเกินสามสิบนาทีแบบไม่มีความคืบหน้า ให้เปลี่ยนไปอ่านข้ออื่นอย่างน้อยห้านาที

3.1.3. Time Complexity

- ก่อนจะเริ่มลงมือเขียนโปรแกรม เราควรจะรู้ให้ได้ว่าสิ่งที่เราเขียนจะได้กี่คะแนนสำหรับปัญหานั้น
- Big O Notation เป็นสิ่งสำคัญที่ควรจะคาดคะเนให้ได้ว่าเราจะได้คะแนนเท่าไร
- จุดอ้างอิงคร่าวๆ ให้คาดว่า C++ สามารถรันได้ $1,000,000,000$ (พันล้าน) คำสั่งต่อหนึ่งวินาที ซึ่งหมายความว่า
  - $N = 1,000$ สามารถคิดวิธี $O(N^3)$ ได้แบบฉิวเฉียด $O(N^2 \log N)$ ได้อย่างปลอดภัย
  - $N = 1,000,000$ ควรจะ $O(N \log N)$
  - $N = 20$ อาจจะเป็น $O(2^N)$
- จำว่า 2^{20} มีค่าประมาณ $1,000,000$ (หนึ่งล้าน)

ขนาด N	เวลาที่ควรใช้
$N \leq 10$	$O(N!)$
$N \leq 20$	$O(2^N)$
$N \leq 500$	$O(N^3)$
$N \leq 5000$	$O(N^2)$
$N \leq 10^6$	$O(N \log N)$ or $O(N)$

ขนาด N	เวลาที่ควรใช้
N is large	$O(1)$ or $O(\log N)$

สัญลักษณ์	ความหมาย
P	ถูกต้อง
-	Incorrect Output
X	Error
T	Timeout

3.2. Big-O Notation

- introduction
- time complexity
- space complexity

3.2.1. การเขียนโปรแกรมแข่งขันประกอบด้วย

- การออกแบบอัลกอริทึม
- การ implement สร้างอัลกอริทึม

3.2.2. การออกแบบอัลกอริทึม

- ใช้ทักษะ problem solving
- และ mathematical thinking
- โดยจำเป็นต้องวิเคราะห์ปัญหาและแก้ปัญหาอย่างสร้างสรรค์
- อัลกอริทึมที่ใช้แก้ปัญหาจะต้องถูกต้องแล้วมีประสิทธิภาพ
- หลักสำคัญของโจทย์ปัญหาส่วนใหญ่จะเป็นการคิดสร้างสรรค์อัลกอริทึมที่มีประสิทธิภาพ
- ความรู้ทางทฤษฎีของอัลกอริทึมเป็นสิ่งสำคัญต่อนักเขียนโปรแกรมเชิงแข่งขัน
- โดยทั่วไปแล้ว เฉลย (solution) ของปัญหาหนึ่งนั้นจะเป็นการใช้เทคนิคที่แพร่หลายร่วมกับ insight ใหม่ตามโจทย์
- เทคนิคในการเขียนโปรแกรมเชิงแข่งขันยังเป็นพื้นฐานในการวิจัยสายอัลกอริทึมอีกด้วย

3.2.3. การ implement สร้างอัลกอริทึม

- ใช้ทักษะการเขียนโปรแกรมที่ดี
- ในการเขียนโปรแกรมเชิงแข่งขันนั้น เฉลยจะถูกตรวจสอบอัลกอริทึมด้วยชุดทดสอบ
- ดังนั้น การที่ไอเดียของอัลกอริทึมถูกต้องเท่านั้นยังไม่เพียงพอ แต่การ implement สร้างอัลกอริทึมจะต้องถูกต้องด้วยเช่นกัน
- style ในการเขียนโค้ดในการแข่งขันจะตรงไปตรงมาและกระทัดรัด
- โปรแกรมควรถูกเขียนอย่างรวดเร็ว เพราะเวลาจะมีอยู่อย่างจำกัด

3.2.4. การวิเคราะห์อัลกอรึทึม

- ปัญหาที่กำหนดให้สามารถแก้ปัญหาด้วยอัลกอริทึมหลายแบบในสภาพเงื่อนไขเดียวกัน
- จำเป็นต้องเรียนรู้วิธีการเปรียบเทียบประสิทธิภาพการทำงานของแต่ละอัลกอริทึม

เพื่อพิจารณาเลือกใช้ได้อย่างถูกวิธี

3.2.5. Motivation problem

- เราจะรู้ได้อย่างไรว่าโค้ดใดนี้มีประสิทธิภาพที่ดีกว่ากัน ระหว่าง

```
for (int i = 1; i ≤ n; i++) {
  for (int j = i + 1; j ≤ n; j++) {
    // code
  }
}
```

- และ

```
for (int i = 1; i ≤ n; i++) {
  for (int j = 1; j ≤ n; j += i) {
    // code
  }
}
```

3.2.6. ประสิทธิภาพของอัลกอริทึม

- สามารถแบ่งได้เป็นสองประเภท คือ
  - การใช้เนื้อที่ในหน่วยความจำ หรือความต้องการในการใช้เนื้อที่ของความจำในการเก็บข้อมูล
  - ประสิทธิภาพของเวลาในการทำงาน หรือระยะเวลาที่ใช้ในการประมวลผลข้อมูล
- โดยทั่วไปแล้วประสิทธิภาพทั้งสองประเภทจะพัฒนาไปพร้อมกันได้ยาก

และส่วนมากจะให้จความสำคัญกับการใช้เวลาอย่างมีประสิทธิภาพมากกว่าเนื้อที่

- **Big O Notation** หรืออันดับขนาด (order of magnitude) เป็นเครื่องมือหลักในการวิเคราะห์อัลกอริทึม

3.2.7. Time complexity

- ประสิทธิภาพของอัลกอริทึมเป็นสิ่งสำคัญในการเขียนโปรแกรมเชิงแข่งขัน
- โดยทั่วไปนั้นจะง่ายที่จะออกแบบอัลกอริทึมที่แก้ปัญหาได้ช้า แต่ความท้าทายจะอยู่ที่อยากออกแบบอัลกอริทึมที่รวดเร็ว
- หากอัลกอริทึมนั้นช้าเกินไป จะได้รับคะแนนบางส่วนหรือไม่ได้เลย
- การคาดการณ์เวลาที่อัลกอริทึมใช้สำหรับ input ต่าง ๆ
- หลักการคือการแสดงประสิทธิภาพในรูปแบบของฟังก์ชันโดยมีพารามิเตอร์เป็นขนาดของ input
- ด้วยการคำนวณ time complexity เราสามารถตรวจสอบความเร็วของอัลกอริทึมได้โดยไม่ต้อง implement ขึ้นมาก่อน

3.2.8. กฎของการคำนวณ

- time complexity ของอัลกอริทึมหนึ่งจะถูกแทนด้วยสัญลักษณ์ $O(\cdots)$ โดยจุดสามจุดแสดงถึงฟังก์ชันใด ๆ
- โดยทั่วไปแล้ว ตัวแปร n แสดงถึงขนาดของ input
- ตัวอย่างเช่น หาก input เป็นอาเรย์ของตัวเลข n จะเป็นขนาดของอาเรย์นั้น และหาก input เป็น string n จะเป็นความยาวของ string นั้น

3.2.9. การวนซ้ำ

- เหตุผลที่พบบ่อยจากการที่อัลกอริทึมช้าเกินไปเกิดจากการใช้การวนซ้ำจำนวนมากสำหรับ input

- ยิ่งวนซ้ำหลายชั้น อัลกอริทึมจะยิ่งช้า
- หากมีการวนซ้ำ k ชั้น time complexity จะเป็น $O(n^k)$

ตัวอย่างเช่น time complexity ของโค้ดด้านล่างจะเป็น $O(n)$

```
for (int i = 1; i ≤ n; i++) {
  // code
}
```

และ time complexity ของโค้ดด้านล่างนี้ $O(n^2)$

```
for (int i = 1; i ≤ n; i++) {     // ชั้นที่1
  for (int j = 1; j ≤ n; j++) {  // ชั้นที่2
    // code
  }
}
```

3.2.10. Order of magnitude

- time complexity จะไม่สนใจจำนวนรอบเป๊ะ ๆ ภายใน loop แต่จะสนใจเฉพาะ order of magnitude
- ในตัวอย่างต่อไปนี้ จำนวนรอบในการวนเป็น $3n$, $n+5$ และ $l\lceil\frac{n}{2}r\lceil$ ครั้ง แต่ time complexity นั้นเท่ากันคือ $O(n)$

```
for (int i = 1; i ≤ 3*n; i++) {
  // code
}
```

```
for (int i = 1; i ≤ n+5; i++) {
  // code
}
```

```
for (int i = 1; i ≤ n; i += 2) {
  // code
}
```

ในอีกตัวอย่างหนึ่ง time complexity ของโค้ดต่อไปนี้เป็น $O(n^2)$

```
for (int i = 1; i ≤ n; i++) {
  for (int j = i+1; j ≤ n; j++) {
   // code
  }
}
```

3.2.11. Phase

- ถ้าอัลกอริทึมประกอบไปด้วยหลาย phase ติดต่อกันแล้ว complexity จะเป็น time complexity ที่สูงสุดของ phase หนึ่ง
- เนื่องจาก bottleneck จะอยู่ที่ phase ที่ช้าที่สุด
- ตัวอย่างเช่น โค้ดต่อไปนี้ประกอบด้วย 3 phases ที่มี time complexity $O(n)$, $O(n^2)$ และ $O(n)$
- ดังนั้นคอขวดจะเป็น $O(n^2)$

```
for (int i = 1; i ≤ n; i++) {
  // code O(n)
}
```

```
for (int i = 1; i ≤ n; i++) {
  for (int j = 1; j ≤ n; j++) {
    // code O(n^2)
  }
}
for (int i = 1; i ≤ n; i++) {
  // code O(n)
}
```

3.2.12. กรณีหลายตัวแปร

- บางครั้ง time complexity จะขึ้นอยู่กับหลายปัจจัย
- ซึ่งเราสามารถใช้หลายตัวแปรแทนได้
- ตัวอย่างเช่น time complexity ของโค้ดด้านล่างเป็น $O(nm)$

```
for (int i = 1; i ≤ n; i++) {
  for (int j = 1; j ≤ m; j++) {
    // code
  }
}
```

3.2.13. Recursion

- time complexity ของฟังก์ชัน recursive จะขึ้นอยู่กับจำนวนครั้งที่ฟังก์ชันนั้นถูกเรียกและ time complexity ของการเรียกฟังก์ชันแต่ละครั้ง
- ซึ่งจะนำสองค่านี้มาคูณกัน
- ยกตัวอย่างเช่น

```
void f(int n) {
  if (n == 1) return;
  f(n-1);
}
```

- การเรียกฟังก์ชัน $f(n)$ จะเรียกฟังก์ชันทั้งหมด n ครั้ง และแต่ละครั้งจะใช้ time complexity $O(1)$
- ดังนั้น time complexity รวมคือ $O(n) \cdot O(1) = O(n)$
- อีกตัวอย่าง เช่น

```
void g(int n) {
  if (n == 1) return;
  g(n-1);
  g(n-1);
}
```

- ในกรณีนี้ การเรียกฟังก์ชั้นแต่ละครั้งจะเรียกเพิ่มอีกสองครั้ง ยกเว้นกรณี $n = 1$
- พิจารณาสิ่งที่เกิดขึ้นจากการเรียก g ด้วยพารามิเตอร์ n
- ตารางด้านล่างแสดงจำนวนการเรียก function ที่เกิดขึ้น

function call	number of calls
$g(n)$	1
$g(n-1)$	2

function call	number of calls
$g(n-2)$	4
...	...
$g(1)$	2^{n-1}

- จากตารางนี้ time complexity คือ $1+2+4+\cdots+2^{n-1}=2^n-1=O(2^n)$

3.2.14. Complexity classes

- รายการต่อไปนี้รวม time complexity ของอัลกอริทึมที่เจอบ่อย
- $O(1)$ constant-time algorithm
  - เวลาที่ใช้ในการรันจะเป็น constant ซึ่งไม่สนใจขนาดของ input
  - ยกตัวอย่างเช่นอัลกอริทึมที่มีสูตรคำนวณคำตอบโดยไม่ต้องทำซ้ำใด ๆ
- $O(\log n)$ logarithmic algorithm
  - อัลกอริทึมแบบ **logarithmic** ส่วนใหญ่จะหารครึ่งขนาดของ input ที่จำเป็นต้องพิจารณาในแต่ละขั้นตอน
  - เวลาที่ใช้ในการรันจะเป็น logarithmic เนื่องจาก $\log_2 n$ เท่ากับจำนวนครั้งที่หาร n ด้วย 2 จนกว่าจะเป็น 1
- $O(\sqrt{n})$ square root algorithm
  - ช้ากว่า $O(\log n)$ แต่เร็วกว่า $O(n)$
  - เป็นกรณีที่หารขนาด input n ด้วย $\sqrt{n}=\frac{n}{\sqrt{n}}$
- $O(n)$ linear algorithm
  - อัลกอริทึมแบบ **linear** จะวนใน input เป็นจำนวนครั้งที่แน่นอน
  - ส่วนใหญ่แล้วจะเป็น time complexity ที่ค่อนข้างดี เนื่องจากปกติแล้วจะจำเป็นต้องอ่านค่าทุกค่าใน input
- $O(n \log n)$
  - ส่วนใหญ่จะเกิดจากการเรียงข้อมูล
  - หรือ data structure ที่ใช้ $O(\log n)$ ในแต่ละ operation
- $O(n^2)$ quadratic algorithm
  - ส่วนใหญ่ใช้ลูปสองชั้น
  - เช่นการพิจารณาทุกคู่ของ input
- $O(n^3)$ cubic algorithm
  - ลูปสามชั้น
- $O(2^n)$
  - การพิจารณาทุก subsets ของ input
  - ตัวอย่างเช่น subsets ของ $\{1,2,3\}$ คือ

  $\emptyset$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$ และ $\{1,2,3\}$.
- $O(n!)$
  - การพิจารณาทุกการจัดเรียงของ input
  - ตัวอย่างเช่นการจัดเรียงของ $\{1,2,3\}$ คือ

  $(1,2,3)$, $(1,3,2)$, $(2,1,3)$, $(2,3,1)$, $(3,1,2)$ and $(3,2,1)$.

3.2.15. การประเมินประสิทธิภาพ

- จากการคำนวณ time complexity ของอัลกอริทึมนั้นทำให้สามารถตรวจสอบประสิทธิภาพของอัลกอริทึมก่อนการ implement ได้
- จุดอ้างอิงคร่าวๆ ให้คาดว่า C++ สามารถรันได้ $1,000,000,000$ (พันล้าน) คำสั่งต่อหนึ่งวินาที
- ตัวอย่างเช่นถ้าข้อจำนวนของโจทย์นั้นให้เวลาในการคำนวณหนึ่งวินาทีและ $n = 10^5$
- ถ้า time complexity เป็น $O(n^2)$ อัลกอริทึมนี้จะรันทั้งหมด $(10^5)^2 = 10^{\{10\}}$ คำสั่ง
- ซึ่งโดยคร่าวใช้เวลาประมาณสิบวินาที ซึ่งจะช้าเกินไปสำหรับโจทย์ข้อนี้
- ในอีกทางหนึ่ง หากว่าเรารู้ขนาด input แล้ว เราสามารถเดา time complexity ของอัลกอริทึมที่ใช้แก้ปัญหาได้เช่นกัน
- ตารางต่อไปนี้บอกการประเมินที่พบบ่อยในกรณีที่ให้เวลาในการคำนวณหนึ่งวินาที
- ยกตัวอย่างเช่น หากขนาดของ input เป็น $n = 10^5$ time complexity ที่ต้องการจากอัลกอริทึมก็คือ $O(n)$ หรือ $O(n \log n)$
- ด้วยความรู้นี้จะทำให้คิดอัลกอริทึมได้ง่ายขึ้นจากการตัดอัลกอริทึมที่ใช้เวลานานออก

input size	required time complexity
$N \leq 10$	$O(N!)$
$N \leq 20$	$O(2^N)$
$N \leq 500$	$O(N^3)$
$N \leq 5000$	$O(N^2)$
$N \leq 10^6$	$O(N \log N)$ or $O(N)$
N is large	$O(1)$ or $O(\log N)$

3.2.16. ตัวแปร constant

- อย่างไรก็ตาม ควรจำว่า time complexity นั้นแค่เป็นการประมาณประสิทธิภาพเท่านั้นเพราะไม่คิด **constant factors**
- ตัวอย่างเช่น อัลกอริทึมที่ใช้เวลา $O(n)$ อาจมีการคำนวณทั้งหมด $\frac{n}{2}$ หรือ $5n$ คำสั่ง
- ซึ่งเป็นสิ่งสำคัญต่อเวลาที่ใช้จริงในการรันอัลกอริทึม

3.2.17. space complexity

- ใช้ Big O Notation เป็นเครื่องมือเช่นกัน
- เปลี่ยนจากการนับขนาดเวลา เป็นนับขนาด memory เช่น
  - ขนาดของ array ที่เราประกาศ

3.3. Stack & Queue

3.3.1. Stack

- เป็นโครงสร้างข้อมูลเชิงเส้น โดยข้อมูลสามารถเข้าและออกได้เพียงทางเดียวที่เรียกว่า **top**
- stack (กองซ้อน) จะใช้หลักเกณฑ์ LIFO (Last In First Out) ก็คือข้อมูลที่ถูกเพิ่มทีหลังสุดจะเป็นข้อมูลที่จะถูกนำออกก่อน
- การเพิ่มข้อมูลลงใน stack จะเรียกว่าการ **push** ส่วนการลบข้อมูลจะเรียกว่าการ **pop**
- ใน stack เราจะมีตัวแปรเก็บตำแหน่งข้อมูลที่เข้าหลังสุดอยู่เสมอ โดยจะเรียกว่า **top**

https://media.geeksforgeeks.org/wp-content/uploads/geek-stack-1.png°

3.3.2. ตัวอย่างการใช้ vector เพื่อ implement stack

```cpp
#include <bits/stdc++.h>

using namespace std;

void printvec(vector<int> &a) {
  for (auto x : a)
    printf("%d ", x);
  printf("\n");
}

int main() {
  // stack
  vector<int> a;
  a.push_back(1);
  a.push_back(2);
  a.push_back(3);
  a.push_back(4);
  printvec(a); // 1 2 3 4
  a.pop_back();
  a.pop_back();
  printvec(a); // 1 2
}
```

3.3.3. Queue

- queue (แถวคอย) เป็นโครงสร้างข้อมูลเชิงเส้น โดยข้อมูลจะเข้าได้ทางหนึ่งที่เรียกว่า **rear** และจะถูกนำออกได้ในอีกทางที่เรียกว่า **front**
- queue จะหลักกร (First In First Out) ก็คือข้อมูลที่ถูกนำเข้าแรกสุดจะเป็นข้อมูลที่ถูกนำออกแรกสุด
- การเพิ่มข้อมูลเข้า queue จะเรียกว่า **enqueue** และการลบข้อมูลจะเรียกว่า **dequeue**
- ใน queue นั้นเราจะมีตัวแปรเก็บตำแหน่งสองตำแหน่ง
  - **front** ในการเก็บข้อมูลปัจจุบันที่ถูกนำเข้า **แรก** สุด
  - **rear** ในการเก็บข้อมูลปัจจุบันที่ถูกนำเข้า **หลัง** สุด

https://media.geeksforgeeks.org/wp-content/uploads/geek-queue-1.png°

3.3.4. Stack vs Queue

Stack	Queue
LIFO	FIFO
one pointer (top)	two pointers (front, rear)
push	enqueue
pop	dequeue
recursion	sequential

3.3.5. Deque (double-ended queue)

- ใน STL จะมีโครงสร้างข้อมูลชื่อ deque โดยจะเป็นโครงสร้างข้อมูลเชิงเส้นที่สามารถเพิ่มลบข้อมูลได้ทั้งสองทิศทางด้วยคำสั่ง
  - push_front
  - push_back
  - pop_front
  - pop_back

3.3.6. ตัวอย่าง deque

```cpp
#include <bits/stdc++.h>

using namespace std;

void printq(deque<int> &a) {
  for (auto x : a)
    printf("%d ", x);
  printf("\n");
}

int main() {
  // double-ended queue
  deque<int> b;
  b.push_back(1);
  b.push_back(2);
  b.push_back(3);
  b.push_back(4);
  printq(b); // 1 2 3 4
  b.pop_front();
  b.pop_front();
  printq(b); // 3 4
  b.pop_back();
  printq(b); // 3
}
```

3.4. Linked List

3.4.1. คำอธิบาย

- linked list เป็นโครงสร้างข้อมูลเชิงเส้น โดยข้อมูลนั้นไม่จำเป็นต้องอาศัยอยู่ในหน่วยความจำที่ตำแหน่งติดกัน
- ข้อมูลใน linked list ถูกเชื่อมต่อกันด้วย pointers ดังรูปด้านล่าง

https://upload.wikimedia.org/wikipedia/commons/6/6d/Singly-linked-list.svg°

- จากรูปจะมีข้อมูลสามชุด ซึ่งเราเรียกว่า **node**
  - ใน node แรกจะเก็บเลข 12 และ pointer ที่ชี้ node ถัดไป
  - node ที่สองเก็บเลข 99 และ pointer ที่ชี้ node ถัดไป
  - node ที่สามเก็บเลข 37 และ pointer ซึ่งยังไม่ได้ชี้ node อื่น

3.4.2. Singly Linked List

- เป็น linked list ประเภทที่ง่ายที่สุด

- โดยหนึ่ง node จะเก็บข้อมูล และ pointer เพื่อชี้ตำแหน่งของ node ถัดไป

3.4.3. ตัวอย่างการ implement singly linked list

- โดยสร้าง struct Node เพื่อเก็บข้อมูล

```
// A simple CPP program to introduce
// a linked list
#include <bits/stdc++.h>
using namespace std;

// A linked list node
struct Node {
  int data;
  struct Node *next;
};

int main() {
  // Program to create a simple linked
  // list with 3 nodes
  Node *head = NULL;
  Node *second = NULL;
  Node *third = NULL;

  // allocate 3 nodes in the heap
  head = new Node();
  second = new Node();
  third = new Node();

  /* Three blocks have been allocated dynamically.
     We have pointers to these three blocks as head,
     second and third
     head         second         third
     |             |             |
     |             |             |
     +---+---+   +---+---+   +---+---+
     | #  |  # |   | #  | #  |   | #  | #  |
     +---+---+   +---+---+   +---+---+

  # represents any random value.
  Data is random because we haven't assigned
  anything yet */

  head->data = 1;      // assign data in first node
  head->next = second; // Link first node with
  // the second node

  /* data has been assigned to the data part of first
     block (block pointed by the head). And next
     pointer of the first block points to second.
     So they both are linked.

     head         second         third
     |             |             |
     |             |             |
     +---+---+     +---+---+   +---+---+
     | 1 | o----> | #  | #  |   | #   | #  |
```

```
    +---+---+     +---+---+---+   +---+---+---+
  */

  // assign data to second node
  second->data = 2;

  // Link second node with the third node
  second->next = third;

  /* data has been assigned to the data part of the second
     block (block pointed by second). And next
     pointer of the second block points to the third
     block. So all three blocks are linked.

     head         second         third
     |             |             |
     |             |             |
     +---+---+     +---+---+     +---+---+---+
     | 1 | o-----> | 2 | o-----> | #  | #  |
     +---+---+     +---+---+     +---+---+---+     */

  third->data = 3; // assign data to third node
  third->next = NULL;

  /* data has been assigned to the data part of the third
     block (block pointed by third). And next pointer
     of the third block is made NULL to indicate
     that the linked list is terminated here.

     We have the linked list ready.

     head
     |
     |
     +---+---+     +---+---+     +---+-------+
     | 1 | o-----> | 2 | o-----> | 3  | NULL |
     +---+---+     +---+---+     +---+-------+

     Note that only the head is sufficient to represent
     the whole list. We can traverse the complete
     list by following the next pointers. */

  while (head != NULL) {
    printf("%d ", head->data);
    head = head->next;
  }

  // This code is contributed by rathbhupendra
}
```

3.4.4. Operations

- insert การเพิ่มข้อมูลใน linked list โดย
  - สร้าง node ใหม่ชื่อว่า newNode (37)
  - นำ pointer ของโหนดตำแหน่งก่อนหน้าที่ต้องการจะแทรก (12) ชี้ไปที่ newNode (37)
  - นำ pointer ของ newNode ไปชี้ที่ pointer ของโหนดตำแหน่งก่อนหน้าเคยชี้ (99)

https://upload.wikimedia.org/wikipedia/commons/thumb/4/4b/CPT-LinkedLists-addingnode.svg/474px-CPT-LinkedLists-addingnode.svg.png°

- delete การลบข้อมูลใน linked list โดย
  - นำ pointer ของโหนดตำแหน่งก่อนหน้าที่ต้องการจะลบ (12) ชี้ข้ามไปที่ node ในตำแหน่งถัดไป (37)

https://upload.wikimedia.org/wikipedia/commons/thumb/d/d4/CPT-LinkedLists-deletingnode.svg/380px-CPT-LinkedLists-deletingnode.svg.png°

3.4.5. Doubly Linked List

- เป็น linked list ที่มีความซับซ้อนเพิ่มขึ้นนิดหนึ่งโดย
- node จะเก็บ pointers สองตำแหน่งสำหรับโหนดก่อนหน้า **prev** และโหนดถัดไป **next**

https://upload.wikimedia.org/wikipedia/commons/thumb/5/5e/Doubly-linked-list.svg/610px-Doubly-linked-list.svg.png°

```c
// linked.c
#include <bits/stdc++.h>
using namespace std;

// what a node look like internally
struct Node {
  int idx;
  struct Node *r_next;
  struct Node *r_prev;
};

int main() {
  // define 3 nodes
  struct Node n0, n1, n2;

  // initialize the nodes
  n0.idx = 0;       // the first node has an index of 0
  n0.r_next = &n1;  // with a reference to the second node
  n0.r_prev = NULL; // and a reference to NULL b/c there's no previous node

  n1.idx = 1;
  n1.r_next = &n2;
  n1.r_prev = &n0;

  n2.idx = 2;
  n2.r_next = NULL;
  n2.r_prev = &n1;

  // create a new reference to n0 called r_first_element
  struct Node *r_first_element = &n0;
  printf("The first node lives @ %p\n", r_first_element);
  printf("The first node has idx: %d\n", r_first_element->idx);

  // print n1 info
  printf("The second node lives @ %p\n", r_first_element->r_next);
  printf("The second node has idx: %d\n", r_first_element->r_next->idx);
```

```
  // print n2 info
  printf("The third node lives @ %p\n", r_first_element->r_next->r_next);
  printf("The third node has idx: %d\n", r_first_element->r_next->r_next->idx);
}
```

3.5. Dynamic Array

3.5.1. คำอธิบาย

- dynamic array เป็น array ที่สามารถยืดหดขนาดได้
- ใน STL จะมี **vector** เป็น dynamic array ให้ใช้

3.5.2. แนวทาง

- เมื่อต้องการเพิ่มข้อมูลลงในอาเรย์ แต่อาเรย์นั้นเต็มแล้ว ให้สร้างฟังก์ชันโดย
- ฟังก์ชันนั้นสร้างอาเรย์ขนาดสองเท่าจากเดิม และคัดลอกข้อมูลจากอาเรย์เดิมมาใส่

3.5.3. การเพิ่มข้อมูล

- หากมีการขยายขนาดอาเรย์และคัดลอกข้อมูลจะทำให้ใช้เวลา O(n)
- แต่กรณีนี้เกิดขึ้นเมื่ออาเรย์เต็มเท่านั้น
- ดังนั้นในกรณีส่วนมากยังใช้เวลาเป็น O(1) อยู่ดี
- ด้วย amortized analysis สามารถถัวเฉลี่ยการเพิ่มข้อมูลเป็น O(1) ได้

https://media.geeksforgeeks.org/wp-content/uploads/dynamicarray.png°

3.5.4. Delete Element

- การลบข้อมูลสามารถทำได้ตามปกติเหมือนอาเรย์ทั่วไป
- ในทางทฤษฎี สามารถลดขนาดอาเรย์ลงได้ แต่ไม่จำเป็นใน competitive programming

https://media.geeksforgeeks.org/wp-content/uploads/DeleteArray.png°

3.5.5. ตัวอย่าง implementation แบบไม่ใช้ vector

```
#include <bits/stdc++.h>

using namespace std;

int cnt = 0;
int sz = 1;
int *A = new int[sz];

void add(int x) {
  if (cnt == sz) {
    sz *= 2;
    int *tmp = new int[sz];
    for (int i = 0; i < cnt; i++) {
      tmp[i] = A[i];
    }
    free(A);
    A = tmp;
  }
  A[cnt++] = x;
}
```

```
void remove() { cnt--; }

void removeAt(int j) {
  for (int i = j; i < cnt; i++) {
    A[i] = A[i + 1];
  }
  cnt--;
}

void print() {
  printf("sz = %d, cnt = %d\\n", sz, cnt);
  for (int i = 0; i < cnt; i++) {
    printf("%d ", A[i]);
  }
  printf("\\n");
}

int main() {
  add(1);
  print();
  add(2);
  print();
  add(3);
  print();
  add(4);
  add(5);
  print();
  add(6);
  print();
  remove();
  print();
  removeAt(3);
  print();
}
```

3.6. Binary Tree

3.6.1. คำอธิบาย

- tree เป็นโครงสร้างข้อมูลแบบมีชนชั้น ซึ่งแตกต่างจาก array, linked list, stack และ queue ที่เป็นโครงสร้างเชิงเส้น
- ตัวอย่าง tree เบื้องต้นคือ binary tree โดยแต่ละ node สามารถมีลูกได้มากที่สุดไม่เกิน 2 ลูก ดังรูป
- ซึ่งเราสามารถเรียกลูกว่า left และ right child ได้

https://media.geeksforgeeks.org/wp-content/cdn-uploads/binary-tree-to-DLL.png°

3.6.2. คำศัพท์

- node ด้านบนสุดจะถูกเรียกว่า **root**
- ลูกด้านล่างที่อยู่ติดกับ node โดยตรงจะเรียกว่า **children**
- node ด้านบนจะเรียกว่า **parent**
- เช่น `a` เป็น child ของ `f` และ `f` เป็น parent ของ `a`
- และ node ที่ไม่มีลูกเลยจะเรียกว่า **leaves**

```
      tree
      ----
       j    <-- root
     /   \
    f     k
  /   \     \
 a     h     z    <-- leaves
```

3.6.3. เมื่อไรถึงใช้ tree?

- หากต้องการเก็บข้อมูลที่มีลำดับขั้น เช่น ระบบไฟล์ในคอมพิวเตอร์

```
file system
-----------
     /    <-- root
   /      \
 ...       home
         /      \
      ugrad      course
       /      /    |    \
     ...     cs101 cs112 cs113
```

- tree (ประกอบกับ ordering บางอย่าง e.g., BST) สามารถเข้าถึงข้อมูลได้เร็วกว่า linear data structure
- tree สามารถเพิ่มลบข้อมูลได้สะดวกกว่าในบางกรณี
- tree สามารถ implement ในรูปแบบคล้ายกับ linked list ได้

3.6.4. Binary Tree

- คือ tree ที่แต่ละ node มีลูกไม่เกินสองลูก คือ left และ right

3.6.5. ตัวอย่าง

- Binary Tree node มีส่วนประกอบต่อไปนี้
  - ข้อมูล
  - Pointer to left child
  - Pointer to right child

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Node {
  int data;
  struct Node *left;
  struct Node *right;

  // val is the key or the value that
  // has to be added to the data part
  Node(int val) {
    data = val;

    // Left and right child for node
    // will be initialized to null
    left = NULL;
    right = NULL;
  }
```

```
};

int main() {
  /*create root*/
  struct Node *root = new Node(1);
  /* following is the tree after above statement

     1
     / \
     NULL NULL
   */

  root->left = new Node(2);
  root->right = new Node(3);
  /* 2 and 3 become left and right children of 1
     1
     / \\
     2     3
     / \\    /  \\
     NULL NULL NULL NULL
   */

  root->left->left = new Node(4);
  /* 4 becomes left child of 2
        1
     /     \\
     2        3
     / \\     / \\
     4 NULL NULL NULL
     / \\
     NULL NULL
   */

  return 0;
}
```

3.6.6. ใช้ array ในการสร้าง binary tree

- ขนาดเรารู้ขนาดของ binary tree แล้วเราสามารถใช้ array ได้
- Consider node `A[i]` has
  - `A[i*2]` as left child
  - `A[i*2+1]` as right child

3.7. Heap

3.7.1. คำอธิบาย

- Heap เป็นโครงสร้างข้อมูลต้นไม้ (tree) ที่มีเงื่อนไขพิเศษ และเป็น complete binary tree
- โดยทั่วไปแล้ว heap มีสองประเภท
  - Max-Heap: ใน max heap นั้น key ที่ root จะมีค่าที่มากที่สุด และคุณสมบัตินี้เป็นจริงในทุก ๆ sub-trees
  - Min-Heap: ใน min heap นั้น key ที่ root จะมีค่าที่น้อยที่สุด และคุณสมบัตินี้เป็นจริงในทุก ๆ sub-trees

https://www.geeksforgeeks.org/wp-content/uploads/MinHeapAndMaxHeap.png

3.7.2. คุณสมบัติ

Binary Heap เป็น Binary Tree ที่มีคุณสมบัติเพิ่มเติมดังนี้

- เป็น complete tree (ทุกระดับจะมีโหนดเต็มระดับยกเว้นระดับสุดท้ายซึ่งระดับสุดท้ายจะต้องมีโหนดทางซ้ายให้ได้มากที่สุด
- คุณสมบัตินี้ทำให้ Binary Heap เหมาะกับการใช้ array ในการเก็บ
- Binary Heap นั้นเป็น Min Heap หรือ Max Heap
  - ใน Min Binary Heap นั้น ค่าที่ root จะเป็นค่าที่น้อยที่สุด และคุณสมบัตินี้ต้องเป็นจริงสำหรับทุก node ใน Binary Tree
  - Max Binary Heap นั้นแค่เปลี่ยนจากค่าที่น้อยที่สุดเป็นมากที่สุด

3.7.3. Operations (Min Heap)

- `getMin()` : คืนค่า root ของ Min Heap ได้เลย ซึ่งใช้เวลา $O(1)$
- `extractMin()` : นำค่าที่น้อยที่สุดใน MinHeap ออก ซึ่งใช้เวลา $O(\log\{n\})$ เพราะเมื่อนำ root ออกแล้วจำเป็นต้องปรับโครงสร้างให้ตรงตามคุณสมบัติ heap
- `decreaseKey()` : ลดค่าของโหนดที่ระบุ ซึ่งใช้เวลา $O(\log\{n\})$ เพราะจำเป็นต้องปรับโครงสร้างให้ตรงตามคุณสมบัติ heap
- `insert()` : เพิ่มโหนดใหม่ ซึ่งใช้เวลา $O(\log\{n\})$ เพราะสามารถเพิ่มโหนดที่ตำแหน่งสุดท้าย แล้วค่อยปรับโครงสร้างให้ตรงคุณสมบัติ heap
- `delete()` : ลบโหนด ซึ่งใช้เวลา $O(\log\{n\})$ เพราะสามารถ `decreaseKey()` ให้เป็น $-\inf$ แล้ว `extractMin()`

3.7.4. ตัวอย่างการ implement

- implement subroutine สองฟังก์ชันคือ
  - `jom` เป็นการปรับโหนดตำแหน่งที่ระบุให้จมลงไปตามคุณสมบัติของ heap
  - `loy` เป็นการปรับโหนดตำแหน่งที่ระบุให้ลอยขึ้นตามคุณสมบัติของ heap

```cpp
#include <bits/stdc++.h>
int d[100010] = {};
int cnt = 0, heap[100010] = {};

using namespace std;

void jom(int i) {
  while (i ≤ cnt / 2) {
    int j = i;
    if (i * 2 < cnt)
      if (d[heap[i * 2]] < d[heap[j]])
        j = i * 2;
    if (i * 2 + 1 < cnt)
      if (d[heap[i * 2 + 1]] < d[heap[j]])
        j = i * 2 + 1;
    if (j == i)
      break;
    swap(heap[i], heap[j]);
    i = j;
  }
```

```
}

void loy(int i) {
  while (i > 1) {
    if (d[heap[i / 2]] > d[heap[i]]) {
      swap(heap[i / 2], heap[i]);
      i /= 2;
    } else
      break;
  }
}

void insertKey(int k) {
  // First insert the new key at the end
  d[++cnt] = k;
  heap[cnt] = cnt;

  // Fix the min heap property if it is violated
  loy(cnt);
}

void decreaseKey(int i, int new_val) {
  d[heap[i]] = new_val;
  loy(i);
}

// Method to remove minimum element (or root) from min heap
int extractMin() {
  // Store the minimum value, and remove it from heap
  int root = heap[1];
  heap[1] = heap[cnt];
  cnt--;
  jom(1);
  return d[root];
}

// This function deletes key at index i. It first reduced value to minus
// infinite, then calls extractMin()
void deleteKey(int i) {
  decreaseKey(i, INT_MIN);
  extractMin();
}

int getMin() { return d[heap[1]]; }

void printHeap() {
  for (int i = 1; i ≤ cnt; i++) {
    printf("%d ", d[heap[i]]);
  }
  printf("\n");
}

int main() {
  insertKey(3);
  insertKey(2);
  printHeap();
```

```
  deleteKey(1);
  printHeap();
  insertKey(15);
  insertKey(5);
  insertKey(4);
  insertKey(45);
  printHeap();
  printf("extractMin %d\n", extractMin());
  printHeap();
  printf("getMin %d\n", getMin());
  decreaseKey(2, 1);
  printHeap();
  printf("getMin %d\n", getMin());
  return 0;
}
```

3.8. Priority Queue

3.8.1. คำอธิบาย

- Priority Queue เป็นส่วนเสริมของ queue โดยเพิ่มคุณสมบัติดังนี้
  - ทุก ๆ สมาชิกจะมี priority ของตนเอง
  - สมาชิกที่มี priority สูงสุดจะถูก dequeue ก่อนสมาชิกที่มี priority ต่ำกว่า
  - หากมีสมาชิกที่ priority เท่ากัน จะถูกนำออกตามลำดับเข้าของ queue
- ใน priority queue ด้านล่าง สมาชิกที่มีค่า ASCII มากจะมี priority สูง

https://media.geeksforgeeks.org/wp-content/cdn-uploads/Priority-Queue-min-1024x512.png°

3.8.2. Operations

- `insert(item, priority)` : เพิ่มสมาชิกพร้อมระบุ priority
- `getHighestPriority()` : คืนค่าสมาชิกที่มี priority สูงสุด
- `deleteHighestPriority()` : นำออกสมาชิกที่มี priority สูงสุด

3.8.3. Implementation

- Heap: สามารถใช้ heap ในการสร้าง priority queue ได้
- STL: ใน STL นั้นมี priority_queue ให้ใช้ได้เลย

3.8.4. ตัวอย่าง (STL)

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  priority_queue<int, vector<int>, greater<int>> q;
  q.push(3);
  q.push(2);
  q.push(15);
  q.push(5);
  q.push(4);
  q.push(45);
  printf("top %d\n", q.top());
```

```
  q.pop();
  printf("top %d\n", q.top());
  q.pop();
  printf("top %d\n", q.top());
  return 0;
}
```

3.9. Binary Search Tree

3.9.1. Description

- Binary Search Tree เป็น binary tree ที่มีคุณสมบัติเพิ่มเติม
  - ทุกโหนดใน subtree ด้านซ้ายจะต้องมีค่าน้อยกว่า root
  - ทุกโหนดใน subtree ด้านขวาจะต้องมีค่ามากกว่า root
  - subtree ด้านซ้ายและด้านขวาจะต้องเป็น binary search tree

https://media.geeksforgeeks.org/wp-content/uploads/BSTSearch.png°

3.9.2. Operations

- Search
- Insertion
- Traversal
  - Pre-order
  - In-order
  - Post-order

3.9.3. Example

```
#include <cstdio>
#include <cstring>
int tree[1000];

int insert(int n, int p) {
  if (tree[p] == -1)
    return p;
  if (n < tree[p])
    return insert(n, p * 2 + 1);
  else
    return insert(n, p * 2 + 2);
}

void pre(int p) {
  if (tree[p] == -1)
    return;
  printf("%d\n", tree[p]);
  pre(p * 2 + 1);
  pre(p * 2 + 2);
}

void in(int p) {
  if (tree[p] == -1)
    return;
  in(p * 2 + 1);
  printf("%d\n", tree[p]);
```

```
  in(p * 2 + 2);
}

void pos(int p) {
  if (tree[p] == -1)
    return;
  pos(p * 2 + 1);
  pos(p * 2 + 2);
  printf("%d\n", tree[p]);
}

int main() {
  char a[5];
  scanf("%s", a);
  int x;
  scanf("%d", &x);
  int i;
  for (i = 0; i < 1000; i++)
    tree[i] = -1;
  while (x--) {
    int n;
    scanf("%d", &n);
    if (tree[0] == -1)
      tree[0] = n;
    else
      tree[insert(n, 0)] = n;
  }

  if (!strcmp(a, "PRE"))
    pre(0);
  if (!strcmp(a, "POS"))
    pos(0);
  if (!strcmp(a, "INF"))
    in(0);
  if (!strcmp(a, "BRF"))
    for (i = 0; i < 1000; i++)
      if (tree[i] != -1)
        printf("%d\n", tree[i]);
  return 0;
}
```

3.10. Set & Map

3.10.1. คำอธิบาย

`set` และ `map` ใน STL นั้นมีความคล้ายกันในแง่ที่ทั้งคู่ใช้โครงสร้างข้อมูล Red Black Tree (self balancing BST ประเภทหนึ่ง) โดยใช้เวลาในการ search, insert และ delete เพียง $O(\log n)$

3.10.2. ข้อแตกต่าง

- `set` ใช้เก็บ keys แต่ `map` ใช้เก็บคู่ key-value
- ตัวอย่างเช่น หากเราต้องการแสดงค่าสมาชิกที่แตกต่างกันเท่านั้น เราสามารถใช้ set ได้เนื่องจากต้องการเก็บแค่ key
- แต่ถ้าหากเราต้องการเก็บความถี่ของแต่ละสมาชิกด้วย เราควรใช้ map เพื่อเก็บ key ตามสมาชิก โดย value เป็นความถี่

3.10.3. Set example

```cpp
// CPP program to demonstrate working of set
#include <bits/stdc++.h>
using namespace std;

int main() {
  set<int> s1;
  // self-balancing binary search tree
  // binary search tree ที่มีความสูงไม่เกิน O(log n) เสมอ
  s1.insert(2); // log(n)
  s1.insert(10);
  s1.insert(5);
  s1.insert(3);
  s1.insert(6);

  cout << "Elements in set:\n";
  for (auto it : s1)
    cout << it << " "; // Sorted

  return 0;
}
```

output

```
Elements in set:
2 3 5 6
```

3.10.4. Map example

```cpp
// CPP program to demonstrate working of map
#include <bits/stdc++.h>
using namespace std;

int main() {
  map<int, int> m;
  // คล้ายๆ dictionary ในภาษา python
  // โครงสร้างเดียวกับ set ก็คือ self-balancing binary search tree

  m[1] = 2; // Insertion by indexing

  // Insertion of pair by make_pair
  m.insert(make_pair(8, 5));

  cout << "Elements in m:\n";
  for (auto it : m)
    cout << "[ " << it.first << ", " << it.second << "]\n"; // Sorted

  map<string, int> m2;
  m2["tt"] = 5; // log(n)
  m2["aw"] = 2399;

  cout << "Elements in m2:\n";
  for (auto it : m2)
    cout << "[ " << it.first << ", " << it.second << "]\n"; // Sorted
```

```
    return 0;
}
```

output

```
Elements in m:
[ 1, 2]
[ 8, 5]
Elements in m2:
[ aw, 2399]
[ tt, 5]
```

3.11. Graph Structure

3.11.1. คำอธิบาย

- กราฟเป็นโครงสร้างข้อมูลไม่เชิงเส้น ประกอบด้วย node (vertex) และ edge
- มีนิยามดังนี้

  > A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes

3.11.2. Example

https://www.geeksforgeeks.org/wp-content/uploads/undirectedgraph.png°

- ในกราฟด้านบนประกอบด้วย set of vertices $V = \{0, 1, 2, 3, 4\}$ และ set of edges $E = \{01, 12, 23, 34, 04, 14, 13\}$.
- กราฟสามารถใช้ในการแก้ปัญหาจริงได้หลายปัญหา
- กราฟสามารถใช้แสดงแทนเครือข่าย
- เช่น เส้นทางในเมือง หรือเครือข่ายโทรศัพท์ หรือวงจรไฟฟ้า เป็นต้น
- กราฟถูกใช้ใน social network เช่น LinkedIn และ Facebook
- เช่น ใน Facebook บุคคลเป็น node ที่เก็บข้อมูลไอดี ชื่อ เพศ ต่าง ๆ และเชื่อมกันด้วยเส้นเชื่อมประเภทต่าง ๆ เช่น การเป็นเพื่อน

3.11.3. Type

https://csacademy.com/lesson/introduction_to_graphs/°

- Undirected graph
- Directed graph

3.11.4. Representation

https://csacademy.com/lesson/graph_representation/°

- adjacency matrix
- adjacency list

3.11.5. Traversal

https://usaco.guide/CPH.pdf#page=119°

- graph traversal

3.12. Hash Table

3.12.1. Idea

- hash table เป็น data structure ที่เอาไว้สร้าง dictionary abstract data structure
- hash function

(figure from [https://en.wikipedia.org/wiki/Hash_table°](https:%%

- หัวใจหลักของการทำ hashing คือการกระจาย entries (key/value pairs) ไปไว้ในช่องต่างๆใน array ที่เรากำหนด
- เมื่อเรามี key แล้วกระบวนการนี้ควรจะคิด index ที่ key นี้ควรจะอาศัยอยู่

```
index = f(key, array_size)
```

- ตามปกติแล้วกระบวนการนี้จะมีสองขั้นตอน คือ

```
hash = hashfunc(key) // คิดค่า hash จาก hash function ของ key
index = hash % array_size // mod ค่า hash ด้วย array_size เพื่อป้องกัน index error
```

- ในกรณีนี้ การ hash จะไม่สนใจขนาดของ array แล้วค่อยจำกัดขอบเขตภายหลังด้วยการ modulo (%) ให้ index อยู่ในช่วง $[0, \text{array\_size} - 1]$

3.12.2. Polynomial hashing

- ตัวอย่าง hash function หนึ่งคือ polynomial hashing ซึ่งมีลักษณะดังนี้

$$\left(s[0]A^{\{n-1\}} + s[1]A^{\{n-2\}} + \cdots + s[n-1]A^0\right) \bmod B$$

- โดย $s[0], s[1], \cdots, s[n-1]$ คือค่าของแต่ละตัวอักษรของ s
- และ A, B เป็นค่าคงที่ที่เรากำหนด

3.12.2.1. ตัวอย่าง

- คำว่า ALLEY จะมีค่าดังตาราง

A	L	L	E	Y
65	76	76	69	89

- หากกำหนดให้ A = 3 และ B = 97 จะทำให้ hash value มีค่าคือ

$$\left(65 \times 3^4 + 76 \times 3^3 + 76 \times 3^2 + 69 \times 3^1 + 89 \times 3^0\right) \bmod 97 = 52$$

- เราก็สามารถกำหนดได้ว่าคำว่า ALLEY ควรจะอยู่ช่องที่ 52

3.12.3. ตัวอย่างโปรแกรม

```cpp
#include <bits/stdc++.h>

using namespace std;

const int A = 3;
const int B = 97;

string hash_table[B];
```

```
int hash_function(string s) {
  int sm = 0;
  for (auto x : s) {
    sm *= A;
    sm += int(x);
    sm %= B;
  }
  return sm;
}

int main() {
  string s = "ALLEY";
  printf("Hash value of %s is %d\n", s.c_str(), hash_function(s));
  hash_table[hash_function(s)] = s;
}
```

ได้ output แบบนี้

```
Hash value of ALLEY is 52
```

💡 ข้อสังเกต

- hash function ควรจะเป็น deterministic function ซึ่งหมายความว่าถึงเราจะรันกี่ครั้งก็ควรจะได้ค่าเดิมเสมอ
- ไม่อย่างนั้นเราจะไม่สามารถรู้ได้ว่าคำๆหนึ่งควรจะอยู่ที่ช่องอะไร หากรันสองครั้งแล้วค่าไม่เหมือนกัน
- ตัวอย่างของ nondeterministic function คือการ random ค่าขึ้นมา ทำให้ไม่มีการการันตีว่ารันสองครั้งแล้วจะได้ค่าเดิม

3.12.4. Collision

- เนื่องจากเวลาทำ hashing จะไม่สามารถการันตีได้ว่าค่า hash ของสองคำที่แตกต่างกันจะแตกต่างกันเสมอ
- เช่นหาก hash value ของ “John Smith” กับ “Sandra Dee” เป็นค่าเดียวกันที่ค่า 152 จะเกิดปัญหาว่าไม่สามารถเก็บทั้งสองค่าในช่องเดียวกันได้
- มีวิธีการแก้ปัญหาหลายแบบ ข้อดีข้อเสียแตกต่างกัน

3.12.4.1. Separate Chaining

https://upload.wikimedia.org/wikipedia/commons/thumb/d/d0/Hash_table_5_0_1_1_1_1_1_LL.svg/900px-Hash_table_5_0_1_1_1_1_1_LL.svg.png°

(figure from https://en.wikipedia.org/wiki/Hash_table°)

- ทำให้แต่ละช่องเป็น linked list

3.12.5. Open addressing

https://upload.wikimedia.org/wikipedia/commons/thumb/b/bf/Hash_table_5_0_1_1_1_1_0_SP.svg/760px-Hash_table_5_0_1_1_1_1_0_SP.svg.png°

(figure from https://en.wikipedia.org/wiki/Hash_table°)

- ขยับไปช่องถัดไป

3.12.6. เปรียบเทียบ

- พิจารณาถึงตอนที่เราต้องการตรวจสอบว่ามีข้อมูลดังกล่าวอยู่ในตารางหรือไม่
- พิจารณากรณีที่มีการลบข้อมูลเกิดขึ้น

3.13. review

3.13.1. stack and queue

- stack: first in last out
  - นึกถึงกองจาน
- queue: first in first out
  - คิวทั่วไป

3.13.2. linked list

- เป็น concept ที่เป็นโครงสร้างแนวเดียวกับ array แต่ข้อดีข้อเสียต่างกัน

  https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2013/03/Linkedlist.png°

3.13.3. dynamic array

- ใช้ vector ของ stl

3.13.4. binary tree

https://upload.wikimedia.org/wikipedia/commons/f/f7/Binary_tree.svg°

3.13.5. heap

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/defa0be5-c84c-4d41-8ad7-ff8d1b906ead/Untitled.png°

3.13.6. priority queue

https://cdn.programiz.com/sites/tutorial2program/files/Introduction.png°

3.13.7. binary search tree

https://upload.wikimedia.org/wikipedia/commons/d/da/Binary_search_tree.svg°

3.13.8. set, map

- เหมือน set ทางคณิตศาสตร์
- map
  - เหมือน dictionary ใน Python
  - เสมือน array ที่ index เป็นอะไรก็ได้

3.13.9. graph

- มี cycle
  - tree เป็น graph ที่ไม่มี cycle

3.13.10. hash table

- data structure ที่ใช้บีบข้อมูลให้เล็กลงโดยอาจจะเสียความละเอียดไป
- เช่น บีบ string ให้เหลือแค่ int ผ่าน hash function แต่ก็มีโอกาสชนกันเป็นปัญหา collision

4. ค่ายต่อ ๆ ไป

4.1. Greedy

4.1.1. หลักการ

- วิธีคิดสำหรับปัญหา optimization โดยทั่วไปแล้วจะมีขั้นตอน ซึ่งแต่ละขั้นจะมีตัวเลือกที่แตกต่างกันไป
- สำหรับหลายๆ ปัญหา optimization อาจจะไม่จำเป็นต้องใช้ dynamic programming เพื่อตัดสินตัวเลือกที่ดีที่สุด
- วิธีคิดแบบ greedy จะเลือกตัวเลือกที่ดีที่สุดในขณะนั้นๆ เสมอ โดยหวังว่าการเลือกตัวเลือกที่ดีที่สุดใน local จะนำไปสู่วิธีการแก้ปัญหาที่ดีที่สุดใน global
- วิธีคิดแบบ greedy จะไม่นำไปสู่วิธีการแก้ปัญหาที่ดีที่สุดเสมอไป มีเพียงบางปัญหาเท่านั้นที่สามารถนำไปใช้ได้
- การพิสูจน์ว่าสามารถใช้วิธีคิดแบบ greedy นั้นทำยาก บางครั้งการลองเขียนโค้ดไปเลยจะสะดวกกว่าการพิสูจน์

4.1.2. ปัญหาการทอนเหรียญ

พิจารณาปัญหาที่ต้องการทอนเงินมูลค่า n โดยใช้เหรียญที่มีมูลค่า $\{c_1, c_2, c..., c_k\}$ จำนวนเหรียญที่น้อยที่สุดที่ต้องใช้เป็นเท่าไร?

4.1.2.1. ตัวอย่าง

มูลค่าของเหรียญคือ

$\{1, 2, 5, 10, 20, 50, 100, 200\}$

และ **n** = 520

4.1.2.2. Greedy approach

ตัวอย่างวิธีคิดแบบ greedy คือเลือกเหรียญที่มีมูลค่ามากสุดก่อนเสมอ จนกว่าเราจะสามารถรวมมูลค่าได้ถูกต้อง

คำเตือน วิธีคิดแบบ greedy ไม่จำเป็นจะเป็นวิธีที่ดีที่สุดเสมอไป กรณีนี้ขึ้นอยู่กับประเภทของเหรียญว่าสามารถใช้วิธี greedy ได้หรือไม่

4.1.3. ปัญหาการจัดตารางเวลา

พิจารณาปัญหาที่ต้องการเรียง **n** งานซึ่งมีเวลาเริ่มและเวลาจบแตกต่างกันไป โดยต้องการให้สามารถจัดเรียงงานให้ได้จำนวนมากที่สุด

4.1.3.1. ตัวอย่าง

งาน	เวลาเริ่ม	เวลาจบ
A	1	3
B	2	5
C	3	9
D	6	8

4.1.3.2. คำอธิบาย

แนวคิดที่เลือกงานถัดไปที่เลิกเร็วที่สุดเสมอ สามารถอ้างได้ว่าหากเราเลือกงานที่เลิกช้ากว่างานที่เราเลือก เราจะได้จำนวนงานอย่างมากที่สุดเท่ากับจำนวนงานที่เราเลือกด้วยวิธีเดิม เพราะฉะนั้นการเลือกงานที่เลิกช้ากว่าจะไม่สามารถให้คำตอบที่ดีกว่าได้

4.2. Array Manipulation

4.2.1. คำอธิบาย

Dynamic Range Sum Queries[3]

- Fenwick Tree
  - https://csacademy.com/lesson/fenwick_trees/°
  - https://visualgo.net/en/fenwicktree?slide=1°
- Segment Tree
  - https://csacademy.com/lesson/segment_trees/°
  - https://visualgo.net/en/segmenttree?slide=1°
- Query time vs Preprocess time
- Hash https://visualgo.net/en/hashtable°
- Heap
  - https://csacademy.com/lesson/heaps/°
  - https://csacademy.com/lesson/heap_application_merge_k_sorted_lists/°
  - https://visualgo.net/en/heap?slide=1°
- Union-find Disjoint Sets https://visualgo.net/en/ufds?slide=1°

4.3. Search

4.3.1. หลักการ

- เป็นวิธีทั่วไปที่แก้ปัญหาได้แทบจะทุกปัญหา
- เป็นการ brute force หาทุกคำตอบที่เป็นไปได้

4.3.2. Generating Permutations

เขียน recursive function เพื่อ generate ทุก permutation ที่เป็นไปได้

4.3.3. Graph Traversal

https://visualgo.net/en/dfsbfs?slide=1°

- BFS
- DFS
  - preorder
  - inorder
  - postorder
- Topological Sort

[3]https://cses.fi/problemset/task/1648/°

4.3.4. Binary Search Tree

https://visualgo.net/en/bst?slide=1°

- create
- search(v)
- insert(v)
- remove(v)

4.3.5. Meet in the middle

- เป็นเทคนิคที่ถ้า search จากต้นทางแล้ว space ใหญ่เกินไป ให้ search จากปลายทางแล้วมาเจอกันตรงกลาง

4.4. Dynamic Programming

4.4.1. หลักการ

- Dynamic programming (DP) เป็นเทคนิคที่รวมการค้นหาทั้งหมดเข้ากับวิธีคิดแบบ greedy
- DP จะแก้ปัญหาได้หากปัญหานั้นสามารถแบ่งออกมาเป็นปัญหาย่อยที่ทับซ้อนกันและสามารถแก้ได้อย่างมีประสิทธิภาพ
- ประโยชน์หลักของ DP คือ
  - หาวิธีแก้ปัญหาที่ optimal
  - หาจำนวนวิธีแก้ปัญหา
- ไอเดีย DP เป็นไอเดียที่ง่ายแต่ส่วนที่ยากจะเป็นการนำไปใช้กับปัญหาจริง

ซึ่งต้องใช้ประสบการณ์

- Backtracking สร้างตารางคู่เก็บที่มาขอคำตอบหากโจทย์ต้องการ

4.4.2. ปัญหาการทอนเหรียญ (ปัญหาเดียวกับ Greedy)

พิจารณาปัญหาที่ต้องการทอนเงินมูลค่า n โดยใช้เหรียญที่มีมูลค่า $\{c_1, c_2, \cdots, c_k\}$ จำนวนเหรียญที่น้อยที่สุดที่ต้องใช้เป็นเท่าไร?

หากมูลค่าของเหรียญเป็น

$\{1, 3, 4\}$

มูลค่ารวม n	จำนวนเหรียญที่น้อยที่สุด
0	0
1	1
2	2
3	1
4	1
5	2
6	2
7	2

มูลค่ารวม n	จำนวนเหรียญที่น้อยที่สุด
8	2
9	3
10	3

นิยาม d_n ให้เก็บค่าแทนจำนวนเหรียญที่น้อยที่สุดที่ใช้ทอนเงินมูลค่า n

จะได้ว่า

$d_i = m \in \left\{\left(d_{\{i-1\}} + 1, d_{\{i-3\}} + 1, d_{\{i-4\}} + 1\right)\right\}$

ซึ่งเป็น recursive case โดย base case คือ $d_0 = 0$

4.4.3. Longest Increasing Subsequence

https://cp-algorithms.com/sequences/longest_increasing_subsequence.html°

4.4.4. Review

4.4.4.1. คืออะไร?

- คำนวณค่าใน state หนึ่งจาก state ก่อนหน้า โดยไม่ต้องไปคำนวณใหม่
- recursion ที่เรียก overlapping sub-problem

แล้วเอามาใช้ได้เลยโดยไม่ต้องไปคำนวณใหม่
- แก้โจทย์โดยการจำ เล่นกับหน่วยความจำ เพื่อที่จะได้ไม่ต้องคำนวณซ้ำ
- หาค่าปัจจุบันจากค่าก่อนหน้า แล้วเราหาเคสเล็ก

4.4.4.2. มีอะไรบ้าง?

- quicksum `dp[i] = dp[i-1] + a[i]`
- knapsack `dp[i] = max(dp[i-wi] + ci)`
- fibonacci `dp[i] = dp[i-2] + dp[i-1]`
- dijkstra `dp[v] = min(dp[u] + w)`
- LIS longest increasing subsequence
- pascal (nCr) `dp[i][j] = dp[i-1][j-1] + dp[i-1][j]`
- coin change `dp[i] = f(dp[i-xj])`
- matrix chain N^3, `dp[i][j] = min(f(dp[i][k],dp[k+1][j]))`
- LCS longest common subsequence
- LCA lowest common ancestor
- #paths pascal
- longest common substring lis
- matrix exponential divide&conquer/bitwise
  - x^9
  - $9 = \{1001\}_2$
  - $x^1 * x^8$
- KMP Knutt-Morris-Pratt
- z-function

- floyd-warshall matrix chain N^3, `dp[i][j] = min(f(dp[i][k], dp[k][j]))`
- Manacher algorithm
- TSP travelling salesman $2^n * n$, `dp[i][j]` โดยที่ i เป็น set

ของเมืองที่เคยผ่านไปแล้ว และ j เป็นเมืองสุดท้ายที่อยู่ตอนนั้น
- tribonacci

4.4.4.3. example

4.4.4.3.1. fibonacci

$f(n) = f(n-1) + f(n-2)$
- 1 1 2 3 5 8 11 ..
- $f(10) = f(9) + f(8)$
- $f(9) = f(8) + f(7)$

`for (i: 2 → n) dp[i] = dp[i-1] + dp[i-2]`

4.4.4.3.2. LIS

หาลำดับย่อยที่ยาวที่สุดที่เป็น increasing subsequence

`1 5 3 2 8 9 3 6 → 1 2 3 6 / 1 2 8 9 / 1 5 8 9 / == 4`

N^2 solution:

define
state: `dp[i] = ความยาวสงสุดของ increasing subsequence เมอพจารณาชองท 0-i แลวลงทายดวยเลข a[i]`

คำตอบสุดท้าย `max(dp[1:N])`

update state: new input: x, พิจารณา $a[j] < x$, แล้วเก็บค่า $\mathrm{dp}[i] = \max(\mathrm{dp}[j] + 1)$

backtrack: + ให้เก็บด้วย j เป็นเท่าไร

```
     0 1 2 3 4 5 6 7 8

a  : 1 5 3 2 8 9 3 6
dp : 0 1 2 2 2 3 4 3 4
bt : 0 0 1 1 1 4 5 4 7
```

$N \log\{N\}$ solution:

define state: `dp[i] = คาทนอยทสดทเปนคาสดทายของ increasing subsequence ความยาว i`

update state: new input: x ไล่ตั้งแต่ `dp[j: 0 → mx]`
- ถ้าค่า dp[j] มากกว่า x และ j มีค่าน้อยที่สุด, `dp[j] = x`,
- แต่ถ้าหาไม่ได้เลย ก็ให้ `dp[mx+1] = x`

คำตอบสุดท้าย mx

```
dp[0] = 0
dp[1] = 1
dp[2] = 2
dp[3] = 3
```

```
dp[4] = 6
dp[5] =
```

4.4.5. เพิ่มเติม

- An Introduction to Dynamic Programming (aquablitz11)[4]
- Dynamic Programming by Aj. Nattee[5]

4.5. Ad-hoc

4.5.1. หลักการ

- โจทย์ประเภทนี้เป็นโจทย์ที่เน้นทักษะการ implementation

4.5.2. String

- Rabin-Karp[6]
- Knuth-Morris-Pratt[7]

4.6. Divide & Conquer

4.6.1. หลักการ

- ประกอบด้วยขั้นตอนหลักสามขั้นตอน
  - Divide แบ่งปัญหาให้เป็นปัญหาเดียวกันที่ขนาดเล็กลง
  - Conquer แก้ปัญหาย่อยๆ ให้ได้
  - Combine

  รวบรวมคำตอบของปัญหาย่อยที่ถูกแก้แล้วมารวมกันเพื่อแก้ปัญหาเดิมที่ใหญ่กว่า
- ปัญหาที่ยังสามารถย่อยต่อเพื่อแก้ปัญหาได้เรียกว่า recursive case
- ปัญหาที่ขนาดเล็กเพียงพอที่จะแก้ปัญหาได้โดยไม่ต้องย่อยต่อเรียกว่า base case
- ปัญหาส่วนใหญ่จะใช้เวลา $O(N^? \log N)$ โดย $\log N$

มาจากการแบ่งปัญหาเป็นครึ่ง ๆ ไปเรื่อย ๆ

4.6.1.1. General Pseudocode

```
DAC(p):
    if is_base_case(p):
        solve(p)
    else:
        divide p into p_1, p_2, ..., p_k
        apply DAC(p_1), DAC(p_2), ..., DAC(p_k)
        combine(DAC(p_1), DAC(p_2), ..., DAC(p_k))
```

4.6.2. Binary Search

การหาจำนวน (หรือคำตอบที่ต้องการ) ในอาเรย์ที่มีการเรียงแล้ว โดยตัวอัลกอริธึมจะตรวจสอบกับตัวเลขตรงกลาง ก่อนที่จะแบ่งอาร์เรย์ออกเป็นสองฝั่งเพื่อหาตัวเลขนั้นอีกครั้ง หากตัวเลขที่กำลัง

[4]http://tcpc.me/2019/01/28/an-introduction-to-dynamic-programming.html°

[5]https://github.com/nattee/ioi-training-note/blob/main/Dynamic%20Programming/01-dynamic%20programming.pdf°

[6]https://cp-algorithms.com/string/rabin-karp.html°

[7]https://cp-algorithms.com/string/prefix-function.html°

หาอยู่มากกว่าตัวเลขตรงกลาง ก็จะไปหาจำนวนนั้นในอาร์เรย์ฝั่งขวา แต่ถ้าหากตัวเลขที่กำลังหาอยู่น้อยกว่าตัวเลขตรงกลาง ก็จะไปหาจำนวนนั้นในอาร์เรย์ฝั่งซ้าย

4.6.2.1. Pattern

```
int lo = 0, hi = N;
while (lo ≤ hi) {
  int mi = (lo + hi) / 2; // ระวัง overflow ตอน lo+hi
  if (A[mi] > target) {
    hi = mi - 1;
  } else {
    lo = mi + 1;
  }
}
```

4.6.2.2. STL

- upper_bound
- lower_bound

4.6.2.3. Binary Search คำตอบ

- ปัญหาบางอย่างสามารถสมมุติคำตอบแล้ว check ได้เร็ว
- หากคำตอบมี pattern เช่น ถ้าค่าน้อยสามารถทำได้แล้วค่าที่มากกว่าจะทำได้ด้วยเสมอ ก็จะสามารถ binary search ได้

4.6.3. More on Binary Search

- สอน Binary Search ฉบับสมบูรณ์โคตรๆ (aquablitz11)[8]
- https://csacademy.com/lesson/binary_search/

4.6.4. Closest Pair

https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/

4.6.5. ปัญหาเลขยกกำลัง

พิจารณาวิธีหาค่าของ A^N หากทำการคูณเลขตรงๆ จะเป็น $O(N)$

ถ้าเราแบ่ง

- Recursive case
  - $A^N = A^{\{\frac{N}{2}\}}c \cdot A^{\{\frac{N}{2}\}}$ **เมื่อ** $N \bmod 2 = 0$
  - $A^N = A^{\{l\lfloor\frac{N}{2}r\rfloor\}}c \cdot A^{\{l\lfloor\frac{N}{2}r\rfloor\}}$ **เมื่อ** $N \bmod 2 = 1$
- Base case
  - $A^N = A$ **เมื่อ** $N = 1$
  - $A^N = 1$ **เมื่อ** $N = 0$

[8] https://aquablitz11.github.io/2019/04/12/complete-bsearch-tutorial.html

4.6.6. จำนวน Fibonacci

จำนวนฟิโบนัชชี (เขียนแทนว่า F_n) คือลำดับจำนวนที่ถูกนิยามจากผลบวกของจำนวนฟิโบนัชชีสองตัวก่อนหน้า โดยกำหนดให้จำนวนฟิโบนัชชีสองตัวแรกเป็น 0 และ 1 ตามลำดับ นั่นคือ

$$F_i = \begin{cases} 1 \text{ if } i = 1 \\ F_{i-1} + F_{i-2} \text{ else} \end{cases} \tag{3}$$

โดยจะสามารถเขียนลำดับได้เป็น

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...$

เพราะฉะนั้นแล้ว base case ก็คือ $0 \leq i \leq 1$ และ recursive case ก็คือ $i \geq 2$

4.6.7. Merge Sort

```
void merge(int array[], int const left, int const mid, int const right) {
  auto const subArrayOne = mid - left + 1;
  auto const subArrayTwo = right - mid;

  // Create temp arrays
  auto *leftArray = new int[subArrayOne], *rightArray = new int[subArrayTwo];

  // Copy data to temp arrays leftArray[] and rightArray[]
  for (auto i = 0; i < subArrayOne; i++)
    leftArray[i] = array[left + i];
  for (auto j = 0; j < subArrayTwo; j++)
    rightArray[j] = array[mid + 1 + j];

  auto indexOfSubArrayOne = 0,   // Initial index of first sub-array
      indexOfSubArrayTwo = 0;    // Initial index of second sub-array
  int indexOfMergedArray = left; // Initial index of merged array

  // Merge the temp arrays back into array[left..right]
  while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {
    if (leftArray[indexOfSubArrayOne] ≤ rightArray[indexOfSubArrayTwo]) {
      array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
      indexOfSubArrayOne++;
    } else {
      array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
      indexOfSubArrayTwo++;
    }
    indexOfMergedArray++;
  }
  // Copy the remaining elements of
  // left[], if there are any
  while (indexOfSubArrayOne < subArrayOne) {
    array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
    indexOfSubArrayOne++;
    indexOfMergedArray++;
  }
  // Copy the remaining elements of
  // right[], if there are any
  while (indexOfSubArrayTwo < subArrayTwo) {
    array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
    indexOfSubArrayTwo++;
```

```
      indexOfMergedArray++;
    }
  }
```

4.6.8. Square Root Decomposition

- https://usaco.guide/plat/sqrt?lang=cpp
- https://cp-algorithms.com/data_structures/sqrt_decomposition.html

4.7. Graph Algorithm

4.7.1. Search in Graph

- Depth-First Search
  - https://grader.mwit.ac.th/problem/vance dfs from all node
- Breadth-First Search
  - https://grader.mwit.ac.th/problem/toi17_wall dfs/bfs โจทย์ซับซ้อน
- Backtracking
  - เป็นการเก็บคำตอบย้อนหลัง หลังจากที่เข้าเงื่อนไขบางอย่างแล้ว
  - https://grader.mwit.ac.th/problem/walking_bot_2
- Branch and Bound (implementation)
  - https://grader.mwit.ac.th/problem/snakeword
- Challenge
  - https://grader.mwit.ac.th/problem/teleport ad-hoc, implementation
- Eulerian
- Hamiltonian

4.8. Graph Applications

4.8.1. Colorings

สามารถใช้การ search ต่าง ๆ เช่น dfs ในการระบายสีโหนดได้

ปัญหาการระบายสีโหนดให้โหนดที่ติดกันมีสีต่างกันเสมอเป็นปัญหาที่พบเจอได้บ่อยในสาขาคอมพิวเตอร์

4.8.2. Connectivity check

DFS แล้วเช็คว่า visit ครบทุกโหนดหรือไม่

4.8.3. Finding cycles

https://visualgo.net/en/cyclefinding?slide=1

ถ้า component มี c โหนด และไม่มี cycle เลยแล้ว component นั้นจะต้องมีเส้นเชื่อมทั้งหมด $c-1$ เส้น และเป็นกราฟต้นไม้

4.8.4. Bipartiteness check

กราฟ bipartite เป็นกราฟเมื่อสามารถใช้สีสองสีในการระบายสีโหนดโดยทุกโหนดที่ติดกันต้องมีสีที่แตกต่างกัน

https://cp-algorithms.com/graph/bipartite-check.html

4.9. Weighted Shortest Path

https://visualgo.net/en/sssp?slide=1°

4.9.1. Dijkstra's Algorithm (implementation)

https://cp-algorithms.com/graph/dijkstra.html°

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}
```

- https://grader.mwit.ac.th/problem/turboprogramming° dijkstra ตรงๆ

```
// credit Mok MWIT29
#include <bits/stdc++.h>
#define ii pair<int, int>
using namespace std;
vector<ii> adj[100100];
int dist[100100];
int main() {
  int N, M, Q;
  scanf("%d %d %d", &N, &M, &Q);
  for (int i = 1, u, v, w; i ≤ M; i++) {
    scanf("%d %d %d", &u, &v, &w);
    adj[u].push_back(make_pair(v, w));
    // adj[v].push_back(make_pair(u, w));
  }
```

```
    for (int i = 1; i ≤ N; i++)
      dist[i] = INT_MAX;
    priority_queue<ii, vector<ii>, greater<ii>> pq;
    dist[1] = 0;
    pq.push(make_pair(0, 1));
    while (!pq.empty()) {
      int d = pq.top().first;
      int n = pq.top().second;
      pq.pop();
      for (auto x : adj[n]) {
        if (dist[x.first] > d + x.second) {
          dist[x.first] = d + x.second;
          pq.push(make_pair(d + x.second, x.first));
        }
      }
    }
    while (Q--) {
      int a;
      scanf("%d", &a);
      if (dist[a] == INT_MAX)
        printf("-1\n");
      else
        printf("%d\n", dist[a]);
    }
  }
```

- https://grader.mwit.ac.th/problem/town° dijkstra ไม่ตรงมาก
- https://grader.mwit.ac.th/problem/followpeatt° dijkstra แบบมีเงื่อนไข
- https://grader.mwit.ac.th/problem/toi14_logistics° dijkstra ซับซ้อนขึ้น

4.9.2. Bellman-Ford

- Negative cycles

```
// credit from https://github.com/Autoratch/practice
#include <bits/stdc++.h>
using namespace std;

int n, m, s, e;
vector<pair<int, pair<int, int>>> adj;
vector<int> dist;

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  cin >> n >> m >> s >> e;

  adj.resize(m);
  dist.assign(n, INT_MAX);

  for (int i = 0; i < m; i++) {
    int a, b, d;
    cin >> a >> b >> d;
    adj[i] = {d, {a, b}};
  }
```

```
  dist[0] = 0;
  for (int i = 0; i < n - 1; i++)
    for (int j = 0; j < m; j++) {
      int a = adj[i].second.first, b = adj[i].second.second, d = adj[i].first;
      if (dist[a] != INT_MAX and dist[a] + d < dist[b])
        dist[b] = dist[a] + d;
    }

  cout << dist[e];
}
```

4.9.3. Floyd-Warshall (implementation)

https://cp-algorithms.com/graph/all-pair-shortest-path-floyd-warshall.html°

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

- https://grader.mwit.ac.th/problem/toi17_1221° all-pair

4.9.4. Johnson (exist)

4.9.5. A* Search

https://www.redblobgames.com/pathfinding/a-star/introduction.html°

4.10. Minimum Spanning Tree

https://visualgo.net/en/mst?slide=1°

4.10.1. Kruskal's (implementation)

- Union-find

```
// credit from https://github.com/Autoratch/practice
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define MOD 1e9 + 7
#define pii pair<int, pair<int, int>>

int n, m;
vector<int> pa;
priority_queue<pii, vector<pii>, greater<pii>> q;

int root(int x) {
  if (pa[x] == x)
    return x;
  else
    return pa[x] = root(pa[x]);
}

int kruskal() {
```

```
  int ans = 0;

  pa.resize(n);
  for (int i = 0; i < n; i++)
    pa[i] = i;

  while (!q.empty()) {
    int w = q.top().first, x = q.top().second.first, y = q.top().second.second;
    q.pop();
    if (root(x) == root(y))
      continue;
    ans += w;
    pa[root(x)] = pa[root(y)];
  }

  return ans;
}

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  cin >> n >> m;

  for (int i = 0; i < m; i++) {
    int a, b, d;
    cin >> a >> b >> d;
    q.push({d, {a, b}});
  }

  cout << kruskal();
}
```

- https://grader.mwit.ac.th/problem/mst° ตรงสุด ๆ

4.10.2. Prim's

```
// credit from https://github.com/Autoratch/practice
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
#define MOD 1e9 + 7

int n, m, ans;
vector<bool> visited;
vector<vector<pair<int, int>>> adj;
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>
    q;

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  cin >> n >> m;

  adj.resize(n);
  visited.resize(n);
```

```
  for (int i = 0; i < m; i++) {
    int a, b, d;
    cin >> a >> b >> d;
    adj[a].push_back({d, b});
    adj[b].push_back({d, a});
  }

  q.push({0, 0});

  while (!q.empty()) {
    int w = q.top().first, p = q.top().second;
    q.pop();
    if (visited[p])
      continue;
    visited[p] = true;
    ans += w;
    for (int i = 0; i < adj[p].size(); i++)
      if (!visited[adj[p][i].second])
        q.push(adj[p][i]);
  }

  cout << ans;
}
```

4.11. Tree algorithms

4.11.1. Diameter

- Algorithm 1
  - ดึง node มั่ว ๆ มาเป็น root
  - หา path ยาวสุดของแต่ละลูก แล้วหาเส้นผ่านศูนย์กลาง
- Algorithm 2
  - เลือกโหนดมั่ว ๆ เป็นโหนด a
  - หาโหนดที่อยู่ไกลที่สุดจากโหนด a เป็นโหนด b
  - หาโหนดที่อยู่ไกลที่สุดจากโหนด b เป็นโหนด c
  - จะได้เส้นผ่านศูนย์กลางคือเส้นทางจาก b ไป c

4.11.2. Lowest common ancestor (implementation)

```
// credit from https://github.com/Autoratch/practice
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5 + 1;

int n, q;
int dp[21][N], lv[N];
vector<int> adj[N];

void dfs(int u, int p, int l) {
  dp[0][u] = p, lv[u] = l;
  for (int v : adj[u])
    dfs(v, u, l + 1);
```

```
}

int lca(int a, int b) {
  if (lv[a] < lv[b])
    swap(a, b);
  for (int i = 20; i ≥ 0; i--)
    if (lv[dp[i][a]] ≥ lv[b])
      a = dp[i][a];
  if (a == b)
    return a;
  for (int i = 20; i ≥ 0; i--)
    if (dp[i][a] ≠ dp[i][b])
      a = dp[i][a], b = dp[i][b];
  return dp[0][a];
}

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  cin >> n;

  for (int i = 0; i < n - 1; i++) {
    int a, b;
    cin >> a >> b;
    adj[a].push_back(b);
  }

  dfs(0, 0, 1);

  for (int i = 1; i ≤ 20; i++)
    for (int j = 1; j ≤ n; j++)
      dp[i][j] = dp[i - 1][dp[i - 1][j]];

  cin >> q;

  while (q--) {
    int a, b;
    cin >> a >> b;
    cout << lca(a, b) << '\n';
  }
}
```

4.11.3. Euler Tour Technique

https://usaco.guide/gold/tree-euler?lang=cpp° https://cses.fi/problemset/task/1137°

4.11.4. Challenge

- https://grader.mwit.ac.th/problem/toi12_weakpoint° one-cycle, dp?
- https://grader.mwit.ac.th/problem/toi14_technology°

4.12. Strong connectivity

https://cp-algorithms.com/graph/strongly-connected-components.html°

4.12.1. Kosaraju's

```cpp
// credit from https://github.com/Autoratch/practice
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5 + 1;

int n, m;
vector<int> adj[N], rev[N];
stack<int> st;
bool visited[N];

void dfs(int u) {
  if (visited[u])
    return;
  visited[u] = true;
  for (int v : adj[u])
    dfs(v);
  st.push(u);
}

void scc(int u) {
  if (visited[u])
    return;
  visited[u] = true;
  cout << u << ' ';
  for (int v : rev[u])
    scc(v);
}

int main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  cin >> n >> m;

  for (int i = 0; i < m; i++) {
    int a, b;
    cin >> a >> b;
    adj[a].push_back(b);
    rev[b].push_back(a);
  }

  for (int i = 1; i ≤ n; i++)
    if (!visited[i])
      dfs(i);

  memset(visited, 0, sizeof visited);

  while (!st.empty()) {
    int u = st.top();
    st.pop();
    if (visited[u])
      continue;
    scc(u);
    cout << '\n';
```

```
    }
}
```

4.12.2. Challenge

- https://grader.mwit.ac.th/problem/walk_around° cycle, union find, reverse query

4.12.3. เพิ่มเติม

- ตะลุยโจทย์ Graph ระดับโหดใน Competitive Programming (aquablitz11)[9]

4.13. Topological sorting (implementation)

DAG - Directed Acyclic Graph

https://cp-algorithms.com/graph/topological-sort.html°

```
int n; // number of vertices
vector<vector<int>> adj; // adjacency list of graph
vector<bool> visited;
vector<int> ans;

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u])
            dfs(u);
    }
    ans.push_back(v);
}

void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
    reverse(ans.begin(), ans.end());
}
```

- https://grader.mwit.ac.th/problem/toi14_technology°

4.14. Paths and circuits

4.14.1. Eulerian

4.14.1.1. Path

เป็น path ที่ท่องไปในเส้นเชื่อมทุกเส้นเพียงหนึ่งครั้งเท่านั้น

4.14.1.2. Circuit

เป็น Eulerian path ที่เริ่มและจบที่โหนดเดียวกัน

[9] http://tcpc.me/2019/08/19/state-graph-tutorial.html°

4.14.1.3. Existence

เงื่อนไขที่จะมี Eulerian path สำหรับ

- กราฟไม่มีทิศทาง
  - ดีกรีของทุกโหนดเป็นจำนวนคู่ หรือ
  - ดีกรีของสองโหนดเป็นจำนวนคี่ (🧠 เพราะอะไร?)
- กราฟมีทิศทาง
  - ในแต่ละโหนด ดีกรีเข้าจะต้องเท่ากับดีกรีออก หรือ
  - มีโหนดหนึ่งมีดีกรีเข้ามากกว่าดีกรีออกเท่ากับหนึ่ง อีกโหนดหนึ่งมีดีกรีออกมากกว่าดีกรีเข้าเท่ากับหนึ่ง และโหนดที่เหลือมีดีกรีเข้าเท่ากับดีกรีออก

4.14.1.4. Hierholzer's algorithm

4.14.2. Hamiltonian

เป็น path ที่ท่องไปในทุกโหนดเพียงหนึ่งครั้งเท่านั้น

4.14.2.1. Existence

ไม่มี efficient method เนื่องจากยังเป็นปัญหาประเภท NP-hard

5. จิปาถะ

5.1. แนวทางการฝึกด้วยตัวเอง

อ้างอิงเนื้อหาตามลำดับของค่าย สอวน. เพราะเป็นแนวทางหลักในเส้นทางการเขียนโปรแกรมเชิงแข่งขัน

5.1.1. สำหรับก่อน สอวน. ค่าย 1

ถ้าหากไม่เคยเขียนภาษา C++ มาก่อน แนะนำให้เริ่มจากหนังสือ C++ Language Tutorial[10] แล้วใช้หนังสือ Competitive Programmer's Handbook[11]

5.1.2. สำหรับขั้นต่อไป

- หากไม่เข้าใจแล้วอยากอ่านเพิ่มเป็นหัวข้อ
  - USACO Guide[12]
  - CP-Algorithm[13]
- เว็บไซต์สำหรับทำโจทย์ (graders)
  - https://cses.fi/problemset/ แหล่งเดียวกับหนังสือหลัก
  - https://beta.programming.in.th/ เว็บหลักในไทย
  - https://otog.in.th/ เว็บที่นักเรียน สอวน ช่วยกันทำขึ้นมาเอง
  - https://codeforces.com/ เว็บหลักในต่างประเทศ
  - Kattis[14]
  - AtCoder[15] (ใช้ร่วมกับ Kenkoooo[16])
- หนังสือเพิ่มเติม (เนื้อหาจะซ้ำซ้อนกับด้านบน) https://usaco.guide/PAPS.pdf
- การบรรยายวิชา การออกแบบและวิเคราะห์อัลกอริทึม โดย สมชาย ประสิทธิ์จูตระกูล[17]

[10] https://cplusplus.com/files/tutorial.pdf
[11] https://cses.fi/book/book.pdf
[12] https://usaco.guide/
[13] https://cp-algorithms.com/
[14] https://open.kattis.com/
[15] https://atcoder.jp/
[16] https://kenkoooo.com/atcoder#/table/
[17] https://www.youtube.com/watch?v=1S0mP_I8YzU&list=PL0ROnaCzUGB65_YkASLAEmcW_mtxFtq4m